Taxonomy of Software Log Smells

Nyyti Saarimäki[1], Donghwan Shin[2], and Domenico Bianculli[1]

[1]University of Luxembourg, Luxembourg, `firstname.lastname@uni.lu`
[2]University of Sheffield, United Kingdom, `d.shin@sheffield.ac.uk`

Abstract

Background: Logging is an important part of modern software projects; logs are used in several tasks such as debugging and testing. Due to the complex nature of logging, it remains a difficult task with several pitfalls that could have serious consequences. Several other domains of software engineering have mitigated such threats by identifying the early signs of more serious issues, i.e., "smells". However, this concept is not yet properly defined for logging.
Objective: The goal of this study is to create a taxonomy of log smells that can help developers write better logging code. To further help the developers and to identify issues that need more attention from the research community, we also map the identified smells to existing tools addressing them.
Methods: We identified logging issues and tools by conducting a survey of the scientific literature. After extracting relevant data from 45 articles, we used them to define logging issues using open coding technique and classified the defined issues using card sorting. We classify the tools based on their reported output.
Results: The paper presents a taxonomy of ten log smells, describing several facets for each of them. We also review existing tools addressing some of these facets, highlighting the lack of tools addressing some log smells and identifying future research opportunities to close this gap.

1 Introduction

Logging is a critical part of modern software development; it is used in tasks such as anomaly detection, quality assurance, and root cause analysis (Cândido et al., 2021). However, producing good logs is non-trivial; it requires making technical and design decisions as well as coordinating the collaboration among developers. Good quality logging is important since bad logging can cause serious issues and delays in a software project (He et al., 2022).

Due to the importance of logging, a wide variety of issues and practices related to software logging have been studied in the literature. The research commonly divides the different aspects of logging into *what, where, when, and how to log*. For example, Gu et al. (2023) investigated logging practices along with issues related to logging, classified them into the why, where, what and how to log categories, and identified eight high-level logging issues in their systematic mapping study.

This categorization is mainly focused on producing logs, e.g., which aspects of a program should be logged and where in the source code the logging code lines should be placed. Even though such a categorization is needed for the production of logs, it is not ideal for identifying issues and bad practices developers may face when writing or maintaining logging code. In other areas of software engineering (e.g., requirements engineering, coding, and testing), the indicators of deeper design problems, recurring problems, or issues impacting quality are often called *smells* (Sharma and Spinellis, 2018).

Despite software smells being a commonly accepted concept in software engineering, only a few works have studied them in the context of logging. Chen and Jiang (2017) defined logging code smells as "poor design and implementation choices when developing the logging code" while Li et al. (2019) defined duplicate logging code smell as "surface indication that usually corresponds to a deeper problem in the system". While both definitions are similar to the ones presented in other areas of software engineering, neither of the studies focused on the concept itself or the general definition of log smells. The former study investigated only long logging snippets, while the latter focused on the duplicate logging code smell. Therefore, there is a lack of work investigating log smells comprehensively.

The goal of this study is to create a taxonomy of log smells and understand whether the research community has already proposed solutions for either detecting or solving them. The taxonomy will help developers be more aware of logging issues and their consequences. Identifying potential logging issues in an early stage helps avoid more serious issues. Furthermore, mapping available tools to the log smells identified in taxonomy will enable developers to use such tools and avoid the smells in their software. Such a mapping will also pinpoint smells not (fully) addressed by current detection/mitigation tools, providing directions for future research work.

We conducted the study as a literature survey from Google Scholar. We extracted problems related to logging highlighted in the literature and grouped similar problems together using open coding to form logging issues. The issues were categorized into groups by conducting several rounds of card sorts and discussions among the authors. We categorized the tools based on their output. We established a taxonomy comprising ten log smells, which can be further divided into several facets. The process for defining the log smells in the catalogue also revealed five direct causes and four consequences for them. We identified eight tools that detect different facets from seven log smells and 13 tools that address (and repair) facets from six log smells.

To summarize, the main contributions of this article are:

- The definition of a log smell taxonomy containing ten log smells and their facets.
- Mapping log smells and their facets to tools detecting and solving them.

The rest of the paper is structured as follows. Section 2 provides background information on logging. Section 3 introduces the concept of log smell. Section 4 illustrates the study design. Section 5 presents the results, while Section 6 discusses them. Section 7 discusses the threats to validity. Section 8 provides an overview of the related work. Finally, Section 9 concludes the paper.

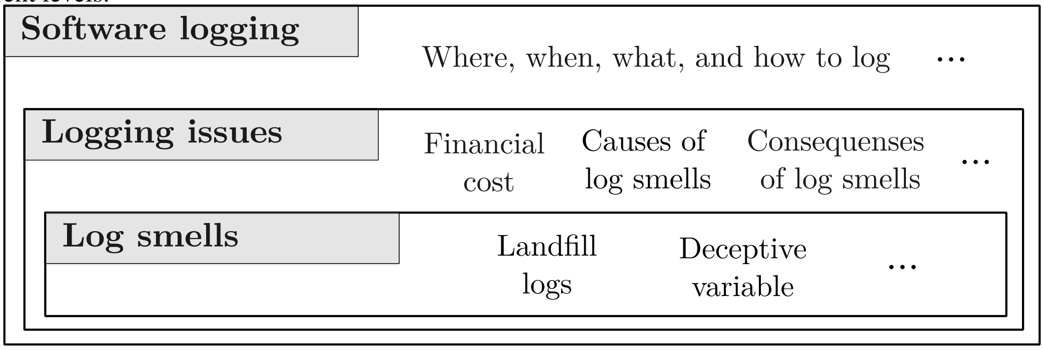
2 Preliminaries

A *log* is a sequence of logging entries usually saved in a text file called *log file*. A *log entry* is either a single line or multiple contiguous lines of text consisting of a (single) header and a message. The *header* provides the developer meta-knowledge about the entry, such as the time the logging code was executed (timestamp), the identifier for the process or component that generated the log entry (process id), and the severity of the logged event (logging level). The header format is typically defined by the underlying logging framework. The *log message* is the part of a log entry presenting information about the state of the software system at runtime. It consists of a static part (often called *log (message) template*) corresponding to the predefined text written by the developer as part of a logging statement, and a dynamic part, which represents the values of the variables captured in the log message at a certain moment of the software execution. An example of a log entry highlighting its different elements is shown below:

$$\underbrace{\texttt{2024-08-05 11:54:12}}_{\text{Timestamp}}\ \underbrace{\texttt{ERROR}}_{\text{Logging level}}\ \underbrace{\texttt{[database]}}_{\text{Process id}}\ \underbrace{\texttt{Failed to connect to database:}}_{\text{Static part}}$$

$$\hookrightarrow \underbrace{\texttt{Connection timeout}}_{\text{Dynamic part (variable)}}\underbrace{\texttt{. Abort proc.}}_{\text{Static part}}$$

Figure 1: The position of log smells in the context of software logging and examples of topics relevant for different levels.



Log entries (and the log file) are produced by *logging code*. An example of logging code generating the example log entry shown above is given in Listing 1. Often, the code is tangled with the feature code and, hence, scattered across the whole project (Kiczales et al., 1997). The logs and logging code have a tight relationship as the logging code produces the logs. Therefore, some characteristics between the two are shared while others are individual, and the logging and feature code are liable to similar issues. For example, if the logging code has an incorrect log message, this is reflected also in the log entries produced by the corresponding statement. However, if the logging code statement is unreasonably long or difficult to understand, that is a problem only in the code.

Listing 1: An example of logging code.

```
try:
    logger.info(f'Connecting to database')
    conn = sqlite3.connect(db_file)
except sqlite3.Error as e:
    logger.error(f'Failed to connect to database: {e}. Abort proc.')
```

Logs can be inspected differently depending on whether the source code producing the logs can be accessed. A *black-box setting* occurs when only the log files are accessible. Consequently, it is impossible to exactly know how and where the log entries were produced. In such a case, having "good quality" log files is crucial. On the other hand, in a *white-box setting*, the source code is available in addition to the log files. In this case, it is possible to understand which part of the source code produced which log entry.

3 Log Smells

Software logging aims to create logs that can be used by developers to debug and monitor the system. It is a process containing several layers; Figure 1 depicts the ones relevant to this paper.

Logging requires actions and decisions from the developers, including the where, when, what, and how aspects, not only when planning to integrate logging into a system but also during the logging process. The quality of logging depends on how these aspects are implemented in practice.

Like any part of software, logging can have different types of issues. Some of them are related to the nature of logging and, therefore, cannot be avoided, while some are related to implementation of logging. For example, logging inherently comes with a financial cost as developing and maintaining it requires resources, while logging too much could be avoided. This paper focuses only on a subset of logging issues called *log smells*.

To the best of our knowledge, log smells have been defined in two previous works. Li et al. (2019) studied duplicate logging code smells and defined them as "surface indication that usually corresponds to a deeper problem in the system". Despite the general nature of the definition, the study does not define log smells in general; instead it focuses on a specific log smell, i.e., log lines having the same text message. The other work adopting the term log smell is by Chen and Jiang (2017). They defined log smells as "poor design and implementation choices when developing logging code". However, this definition limits log smells in the logging code and does not consider the dual nature of logging. In addition to using the term log smell, Chen and Jiang (2017) used the term anti-patterns in logging code to describe "recurrent mistakes which may hinder the understanding and maintainability of the logs". Even though the terms smell and anti-pattern are sometimes used interchangeably, they have distinctive meanings. Anti-patterns are considered to be common but poor solutions to a design or implementation problem that result in issues, such as smells (Brown et al., 1998).

In this paper, we define a log smell as follows:

> Log smell is a poor design choice or an issue impacting quality that could result in a more serious problem, affecting the logging code, the log file, or both.

The definition has two main aspects. First, a smell is an indication of a deeper problem but is not a serious problem itself. For example, a missing logging statement in a catch-block does not prevent the system from running; however, when looking at the code, it is usually clear that a logging statement is missing. Second, log smells do not affect only the logging code but also the log files. This is a natural consequence of the fact that the logs are generated by the logging code. However, the two are not the same and not all issues affecting the code affect the log files and vice versa. The distinction between the two is especially important, to deal with cases in which the engineer reading the logs does not have access to the code that created the log or documentation of the used logging practices.

Each log smell can be manifested in several different ways. In this paper, we refer to them using the term *facet*. For example, having wrong static messages and messages in which the grammar is incorrect both affect the quality of static messages, but they create an issue in different ways. Therefore, they would be considered as facets, while in general having issues with static messages would be a log smell.

4 Study Design

4.1 Goal and Research Questions

The goal of this study is to define and identify software log smells as well as determine the tools that detect, remove, or repair them. Based on this goal, we define the following research questions:

RQ1. *What logging issues are log smells or directly related to them?*

RQ2. *Which log smells have tools or other automated techniques able to detect them?*

RQ3. *Which log smells have tools or other automated techniques able to remove or repair them?*

Although the literature has investigated the wide variety of issues related to logging, to the best of our knowledge no previous work has provided a general definition for log smells or a comprehensive taxonomy of logging issues that could be classified as log smells. We aim to fill this gap by answering RQ1, providing a taxonomy of log smells.

The answers to RQ2 and RQ3 aim to understand which log smells have a tool or other automated solution able to detect a certain smell (RQ2) and remove/repair it (R3). By mapping the tools identified in the literature

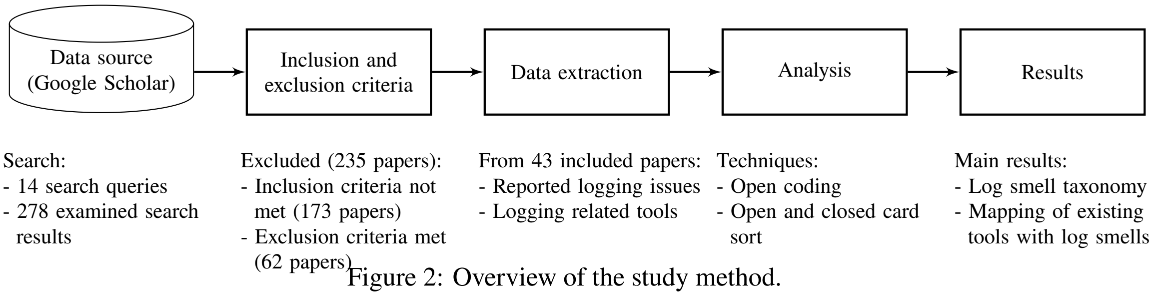


Figure 2: Overview of the study method.

survey with different aspects of logging issues, we can, on one hand, provide practitioners with a catalogue of tools for detecting/removing/repairing log smells; on the other hand, we can identify gaps to be addressed in future research work.

4.2 Methodology

We answered the RQs by conducting a survey on papers published about software logging. We conducted a survey instead of a systematic literature review (SLR) as a proper SLR would have required more resources than the authors had available. The first author conducted the whole survey, and all authors discussed it during the process to make it as unbiased as possible. Despite having limited resources, we aimed to construct and report the survey following best practices and guidelines of SLRs whenever possible (Kitchenham et al., 2023). Figure 2 presents an overview of the survey method.

4.2.1 Paper Selection

Information sources. We conducted the survey using Google Scholar.

Eligibility criteria. Table 1 presents the eligibility criteria applied in this work. We included peer-reviewed papers written in English related to software logging. Additionally, we required the papers to be published in 2000 or after, as we considered papers published before that too old and outdated. We also excluded papers which had an extension paper among the other included papers and kept only the most recently published version of a paper. Finally, papers that did not describe any kind of issues related to software logging were excluded.

Table 1: Inclusion (I) and exclusion (E) criteria of the survey.

ID	Criteria
I1	Written in English
I2	Published after 2000
I3	Related to software logging
I4	Peer reviewed
E1	Variations and versions of the same paper
E2	Not presenting an issue in software logging

Search strategy. Table 2 presents the search strings and the number of search results Google Scholar (GS) reported for them. As some of the search strings resulted in hundreds of results, it was not possible to systematically list and assess all of them. In such cases, the first author (who conducted the review) listed the results from the first five pages of GS's results, i.e., 50 results, and browsed through the remaining pages. This

Table 2: Search strings and number of results. Note that some papers were included in the results of several queries.

Search String	**# Results**	**# Included**
software AND (“log issues” OR“log issue”)	684*	3
“log quality” software	2,840*	7
“logging quality” software	249*	14
“software log quality”	0	0
“software engineering” AND (“logging issues” OR “logging issue”)	132	11
“log smell”	9	0
“log smell” software	1	0
“logging smell” OR “logging smells”	3	
“logging code smell”	5	3
“log anti-pattern” OR “log anti-patterns”	1	1
“logging anti-pattern” OR “logging anti-patterns”	26	12
“bad logging practices” software	16	4
“bad logging practice” software	10	5
“logging bad practices” software	0	0

* results after the first five pages (≈50 results) were only browsed

explains the large difference between the number of results (# Results) and the number of papers included (# Included) for some queries. In case a paper title and abstract looked relevant, the paper was properly assessed (full paper reading). We did not snowball the results as the goal of the study was not to conduct an SLR but to investigate log smells.

Extremely general search terms resulting in millions of search results, such as “quality of log” and “software log issue” were excluded. In such cases, the term “log” often did not mean a software log but could refer to, for example, forestry or a log generated through some other process than software (e.g., oil drilling).

4.2.2 Data Extraction

Selection and data collection process. The data collection and extraction were done between February 2024 and March 2024. We made the searches using the queries and recorded all the results from the Google Scholar searches on a spreadsheet. Listing all results from the searches was needed for the validity of the study, as the search results from Google Scholar could vary over time. During this process, an author labeled each paper relevant or irrelevant based on the title and abstract of the paper. Only papers that were considered relevant after this step were read. The similarity of data extraction was ensured by reassessing all included papers again once the first round of data extraction was completed. Additionally, all authors discussed any issues that arose during the data extraction. Section 7 further discusses potential threats to this process.

The process resulted in 43 papers from which we extracted metadata, reported issues, and software logging-related tools as described in Table 3. The columns “Criteria” and “Description” explain the extracted data, while column “RQ” indicates which research question the extracted data is related to. The metadata was collected from the source of the paper, while the other data was obtained by reading the paper. The issues highlighted in the papers were extracted mostly as direct quotes. Additionally, to separate the different tools in the data and the paper, each tool was given an identifier. If the creators of the tools provided a name, we used that. Otherwise, we identified the tool by using the last name of the first author and the year of publication.

We did not exclude secondary studies, such as SLRs, as the objective was to collect a comprehensive set of issues and tools. We included the issues reported in these papers and cited only the literature review, as

Table 3: Data extraction criteria

Criteria	Description	RQ
Authors	The authors of the paper	Meta
Title	Title of the paper	
Year	Publication year of the paper	
Venue	Publication venue of the paper	
Logging issues	Issues related to logging highlighted by the paper	RQ1
Logging tools	Presented tools related to logging issues	RQ2, RQ3

commonly the authors of the secondary studies had synthesized issues from several papers to form an issue category of their own. This is also consistent with the decision of not snowballing. For the tools reported in secondary studies, we included all the presented tools if relevant and cited the original publications.

4.3 Data Analysis

To answer RQ1, we combined the data extracted from the papers to form general logging issues and qualitatively analyzed them to identify categories suitable for the goals this study. We performed the data analysis in two steps, as illustrated in Figure 3. In step 1, we defined the categories of logging issues while in step 2 we classified the logging issues according to the categories defined in the previous step. The two-step approach allowed us to explore different ways to categorize all potential logging issues initially identified in the data.

We conducted step 1 using several techniques. The first activity in this step was extracting the initial list of issues from the raw data, which was done by one author. We (loosely) aggregated the issues presented in the papers using the *open coding* technique. This is a technique used, for example, in grounded theory studies (Stol et al., 2016) to extract data points from raw data and categorize them. To explore suitable categories for the issues while refining them, we conducted an *open card sort* (Spencer and Garrett, 2009), involving all the authors. In an open card sort, participants receive cards with words on them and are asked to group the cards based on their own criteria; there are no pre-defined groups (i.e., open), and the participants need to define their own categories. Open card sorting can reveal different ways of categorizing the data, as the participants might sort the cards in very different ways. In the card sort conducted for this step, each issue was a single card, and each author did the sort independently using an online platform[1]. Before doing the sort, the authors agreed that our categorization should be relevant to logging issues. For example, an existing categorization of "why, where, what, and how to log" (Gu et al., 2023) would not be suitable for our study, as it is more focused on the process of logging rather than the issues of logging. After conducting the open card sort, the results were discussed among all authors, and the main categories were decided based on the discussion.

First, we slightly refined the issues using open coding and the experience gained from the open card sort conducted in step 1. In step 2, we used similar methods as in step 1. We updated the descriptions of the issues when needed and categorized the issues in the categories defined in step 1 using a *closed card* sort. Unlike an open card sort, a closed card sort begins with a pre-defined set of categories. All authors did the sort and it was conducted similarly as in step 1. The closed card sort resulted in three different classifications of the issues. Therefore, the authors discussed the differences to obtain a consensus. This process was iterative, including further refining the issues and their definitions; and this loop was repeated until the authors reached full agreement. The refinement of issues consisted of combining, renaming, and reorganizing the issues, as well as adding proper descriptions and reasoning why an issue is problematic. During this process we recorded also the facets (i.e. ways an issue can present itself) for the formed logging issues. For example, "access control" and "finding log files" were merged into "log file management". The output of step 2 was

[1] `https://provenbyusers.com/`

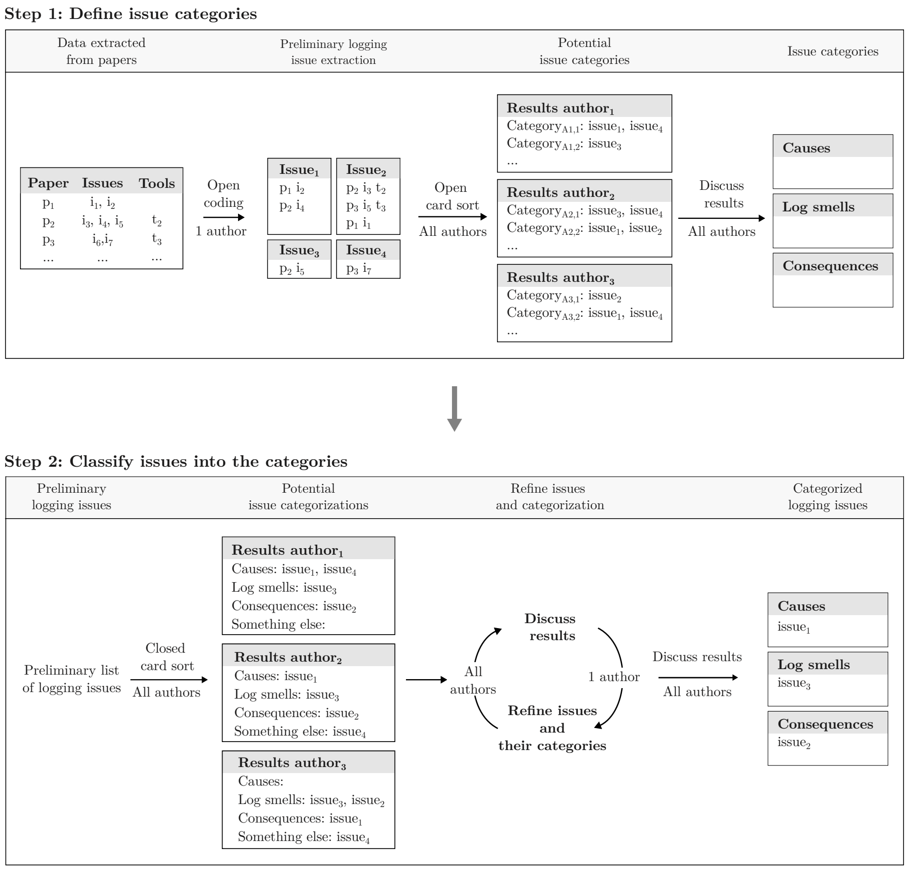


Figure 3: The process for identifying the logging issues and categorizing them.

the final definitions of the logging issues and their categorization.

The data analysis for RQ2 and RQ3 was similar. In RQ2, we identified the tools detecting logging issues, while in RQ3 we focused on tools providing a solution for them. The distinction between the two was made by investigating the output of the tools. If the tool only highlighted an issue, for example, by providing a location in the code, it was considered a detection tool. Solution tools were required to suggest changes or automatically fix issues, such as changing the logging level of a certain log line from error to warning. We analyzed the tools with respect to the log smells defined in the answer to RQ1 to see which smells would be detected or solved. Further, to better understand the aspects covered by the log smells, we mapped the tools to the specific facets of the smells they detected or solved.

4.4 Replicability

To allow replication, verification, and extension of this study, the raw data and complete results of the study will be provided in a replication package. We plan to make it publicly available upon acceptance of the paper.

5 Results

5.1 RQ1: Taxonomy of Log Smells

By answering RQ1, we aim to define a taxonomy of log smells. As discussed in Section 4.3, we went through step 1 and step 2 to analyze the data extracted from the existing papers.

In step 1, we identified 51 logging-related issues. We then categorized the issues by doing an open card sort, which resulted in several different ways of categorization. The proposed categories included, for example, "logging process", "log management", and "dual nature"; the list of issues and the card sort results are provided in the replication package. After discussions among all authors, we decided to use three categories: "Causes of log smells", "Log smells", and "Consequences of log smells". These three categories were determined to be the most appropriate to answer RQ1 as they allow us to identify log smells as well as their causes and consequences, which help better understand their evolution. Specifically, we defined the categories as follows:

- **Causes of log smells**: Issues that can lead to log smells. The issues in the group are not necessarily directly visible in the logging code or log files, but they are aspects of the project that, if not managed properly, will lead to concrete issues or log smells.
- **Log smells**: Issues in the logging code or the actual logs that might cause problems but do not affect the functionality of the project. A more detailed description of log smells is given in Section 3.
- **Consequence of log smells**: Issues created by logging code, log file, or the logging system being affected by log smells over time. We consider only direct issues caused by the log smells and exclude issues that are affected but not directly caused by log smells, such as logs being useless for failure analysis.

In step 2, we categorized the issues into the aforementioned three categories. We refined the description of the issues, but did not change the issues. To make the process complete, we additionally used a fourth category called "Something else" that caught all issues not related to log smells.

The results of the closed card sort indicated a lack of agreement on several issues; therefore, the authors thoroughly discussed the results. This resulted in an iterative process in which the issues were refined and combined, and their classification was reconsidered. The outcome was 19 high-level logging issues which were categorized as log smells, their causes, or consequences. Additionally, we identified seven issues related to other aspects of logging, which are shortly described in A.

Table 4: Overview of categorized logging issues.

Causes of log smells		Log smells		Consequences of log smells	
CA1	Lack of general logging guidelines	LS1	Format turmoil	CO1	Information leak
CA2	Developers' domain knowledge and experience	LS2	Undercover identifier	CO2	Temporal inconsistency
CA3	Logging libraries	LS3	Mercurial logging level	CO3	Effect on system performance
CA4	Separately developed components	LS4	Deceptive variable	CO4	Accidental side-effects of logging code
CA5	Insufficient logging code maintenance	LS5	Variable on the edge		
		LS6	Message madness		
		LS7	Logging lost in the wind		
		LS8	Landfill logs		
		LS9	Sleeping guards		
		LS10	Skeleton in the closet		

An overview of the issues classified in the three categories is presented in Table 4. We describe each of the categories and their associated issues in more detail in the following subsections. To document the issues, we use the template below. However, we have omitted or added elements for some issues based on their relevance.

Issue ID: Name of the issue

- **Description**: General description of the issue and its facets, i.e., the different ways it can be manifested.
- **Implications**: Description of why the issue is problematic and its potential consequences.
- **References**: References to papers mentioning the issue.
- **Example**: Provides an imaginary example of log entries or logging code affected by the issue when such an example is applicable. The problematic parts in the example are highlighted in gray. Additionally, we provide references to real-life examples of the issue presented in other papers.

5.1.1 Causes of log smells

The section presents in detail the five logging issues classified as *causes of log smells*.

CA1: Lack of general logging guidelines

- **Description**: A common cause of log smells is the lack of shared understanding of logging practices among the developers. Each company or developer might have different practices for logging, which creates the need for having rigorous specifications of the logging process and guidelines for logging within a project. Issues related to such guidelines include them being *missing*, *incomplete*, or *outdated*.
- **Implications**: Issues with guidelines can lead developers to adopt different work practices that creates unnecessary inconsistency, confusion, and delays.

- **References**: Anu et al. (2019); Bogatinovski et al. (2022); Gholamian and Ward (2020); He et al. (2018); Li et al. (2020a); Yuan et al. (2012b); Zhu et al. (2015)

CA2: Developer's domain knowledge and experience

- **Description**: Developers writing the logging code make the final decision of what, how, and where to log. However, developers have varying amount of working experience as well as knowledge of different technologies, development methods, and the product being developed. All these factors affect their logging decisions, even if the project has a clear specification of logging. Especially developers not having the *domain knowledge* or not being *familiar with the whole project* might have difficulties making good logging decisions. Knowing the whole system is particularly important when the project consists of several systems, as retrieving meaningful information from the log files can be complicated in such cases.

- **Implications**: Creates unnecessary differences in logging practices and logs.

- **References**: Anu et al. (2019); He et al. (2022); Liu et al. (2020); Rong et al. (2023); Yang et al. (2021); Zhu et al. (2015); Zhao et al. (2017a); Zhaoxue et al. (2021)

CA3: Logging libraries

- **Description**: Adopting a logging library can result in more systematic logs, with developers having several options to choose from. As the libraries and other logging solutions have different functionalities, a project might use *several logging libraries*. However, the libraries might not be able to produce uniform logging because they might not be compatible with each other or have limited configurability. Even if the libraries could be configured similarly, the configurations could be *inconsistent*. Additionally, when new versions of libraries are released, the projects need to update them, which requires resources and might also require changes in the logging code.

- **Implications**: Using multiple logging libraries increases the effort needed for producing systematic and uniform logs throughout a project. The changes required by the updates require extra resources from the development team and might cause unexpected errors.

- **References**: Hassani et al. (2018); Chen and Jiang (2019, 2022)

CA4: Separately developed components

- **Description**: Nowadays, a software project can consist of several components, which all might have been created by different development teams. While this has its advantages, it can introduce complexity in logging. The components might be developed with different technologies which increases the likelihood that they have adopted different libraries for logging. The different components might also use different logging styles, locations, and formats.

- **Implications**: Having several components or concurrency makes using the log files more complex, as the logs from different components might be significantly different. Another consequence is the need for more thorough planning of the project's logging practices.

- **References**: Bijvank et al. (2013); He et al. (2018, 2022); Marron (2018); Yang et al. (2021); Zeng et al. (2019)

CA5: Insufficient logging code maintenance

```
1 2024-07-26 12:39:19Z INFO [database] Connecting to database
2 2024-07-26 12:41:46Z WARN [database] Connection slow, retrying
3 2024-07-26 12:42:16Z ERROR [database] Failed to establish connection: Connection
 ↪  timeout. Abort proc.
...
n 2024-07-26 16:34:12Z INFO [database2] Connecting to database
```

Figure 4: The example log template that, when applicable, is used to illustrate the presented logging issues.

- **Description**: Developers frequently update feature code, but it can be challenging for them to also keep the logging code updated. Moreover, changing the logging code does not always improve it, as the changes might address only a specific use case while ignoring the needs of other parts of the project. Further, the changes might introduce mistakes in the logging code, such as typos, adding wrong variables, or making the code less understandable. Several studies show that logging statements are written and refined in an ad-hoc manner and possibly not until when failures have already happened, and logs would have been needed (Gu et al., 2023).
- **Implications**: Eventually, the lack of maintenance might result in logging code that is obsolete, insufficient, or misleading. Different domains also do maintenance differently; for example, compared to server and desktop apps, mobile apps have less maintenance but their developers more frequently delete logging code (Zeng et al., 2019).
- **References**: He et al. (2018); Chen and Jiang (2019, 2022); Fu et al. (2014); Gu et al. (2023); Li et al. (2020a, 2021a); Shen et al. (2023); Yang et al. (2021); Yuan et al. (2012b); Zeng et al. (2019)

5.1.2 Log smells

Log smells are issues that are poor design choices or impact the quality of logs in a way that could result in a more serious problem (see Section 3 for a more thorough definition of log smells). This section presents 10 logging issues categorized as *log smells*. We illustrate the log smells in practice by showcasing a smelly log for each type. When possible, we modified the log shown in Figure 4 to be affected by a smell. For some smells, the log was not applicable, so a different example is used in those cases. In the modified examples, _ denotes a missing value or log entry, while ↪ denotes a line break.

LS1: Format turmoil

- **Description**: Issues in logging format. Logging format defines a structured way of presenting the information in log files. A well defined format allows developers to effectively search the logs and makes them more machine-friendly. However, a format can have several issues. The facets of the smell can be related to the specification and usage of logging format, such as *inconsistency* and *incompleteness* of the format, or a project *not following a format* or having *several formats*.
- **Implications**: This smell increases the complexity and effort of utilizing the information recorded in the log files, especially when inspecting several log files.
- **References**: Bijvank et al. (2013); Chen and Jiang (2022); He et al. (2022); Marron (2018); Patel et al. (2022); Yang et al. (2021)
- **Example**: The following two log entries come from different systems within the same software project. The entries indicate that the systems use different formats for logging the same event. Additionally,

neither of the entries presents all the necessary information shown in Figure 4: line 1 is missing the verbosity level, while line 8 has an incomplete timestamp. These issues would create difficulties in filtering and matching the information.

```
1 [16:00:34.276Z 08/06] [database] ␣ Connecting to database DB1
...
8 INFO  Jun 08 24  sys2: DB2 connecting to database
```

A real-life example of a formatting issue is presented in Patel et al. (2022).

LS2: Undercover identifier

- **Description**: Issues in the identifier of a log entry. Identifiers are used in logs to mark the component (or, in case of concurrency, the thread) which created the log entry. In order to use the logs effectively, log entries must be linked to the specific components or threads that created the log entry. The facets of the smell are *missing*, *vague*, and *wrong* identifiers.

- **Implications**: Without the link, using the information in the log becomes unnecessarily laborious or even impossible. Having proper identifiers is especially important within a project consisting of several components or using concurrency.

- **References**: Xu et al. (2009); Zhao et al. (2023)

- **Example**: Similarly to the log presented in Figure 4, line 1 correctly indicates that a connection is being established, with the database component producing the log entry. However, the second entry is linked the to server component (highlighted in grey) even though it should be indicating, by replacing `[server]` with `[database]`, that the connection to the database is slow. This could be misleading as a connection to the server could also be slow. Finally, the identifier is missing from line 3, which makes it unclear where the error (i.e., connection timeout) occurred.

```
1 2024-07-26 12:39:19Z INFO [database] Connecting to database
2 2024-07-26 12:41:46Z WARN  [server]  Connection slow, retrying
3 2024-07-26 12:42:16Z ERROR ␣ Failed to establish connection: Connection timeout. Abort proc.
```

A real-life example of a missing identifier is presented in Zhao et al. 2023, Listing 1.

LS3: Mercurial logging level

- **Description**: Issues in logging level. The level is used to indicate the importance of a log entry as well as to categorize and filter the logs. Choosing the correct level is challenging, and as a consequence, changes in them are common. The facets of the smell include *missing*, *incorrect*, and *inconsistent* log level.

  A log level is *missing* when the severity is not indicated. *Incorrect* levels occur when the log level is indicated but it does not describe the recorded event correctly; for example, a log entry describing an expected event is categorized as a error. *Inconsistencies* in the assigned levels mean that the same kind of events have different logging levels in different parts of the code, even though they should have the same level.

- **Implications**: Having to filter log entries by their log level when the latter has issues can complicate the analysis and create confusion. In such cases the retrieved data might be missing relevant events for debugging or add noise to the results.

- **References**: Anu et al. (2019); Bogatinovski et al. (2022); Chen and Jiang (2017); Gholamian and Ward (2020); Hassani et al. (2018); Li et al. (2021c,a, 2017b); Patel et al. (2022); Xu et al. (2009); Yuan et al. (2012b); Zhao et al. (2017a); Zhaoxue et al. (2021)

- **Example**: The example affected by the smell deviates from Figure 4 in several ways. Line 2 reports a slow connection to the database and records it as an ERROR, even though the entry should be classified as a WARNING because the system still tries to establish a connection. The next line reports that the connection eventually timed out; this entry does not have a log level, even though it should be labeled as an ERROR. Finally, line 81 reports the system connecting to the database and labels it as an ERROR; however, this is inconsistent since the same message was labeled as an INFO level message on line 1.

```
1 2024-07-26 12:39:19Z INFO [database] Connecting to database
2 2024-07-26 12:41:46Z ERROR [database] Connection slow, retrying
3 2024-07-26 12:42:16Z ␣ [database] Failed to establish connection: Connection timeout. Abort
 ↪ proc.
...
81 2024-07-26 16:34:12Z DEBUG [database2] Connecting to database
```

Real-life examples are provided, among others, in Chen and Jiang 2017, Fig 4; Patel et al. 2022, Tab 15; Hassani et al. 2018, p. 3262; Bogatinovski et al. 2022, Fig 1b; and Li et al. 2017b, p. 1842.

LS4: Deceptive variable

- **Description**: Variables are used in logging to record the initial state of the program at the time of executing the logging code. The facets of this smell are *missing*, *wrong*, *malformed output*, and *varying granularity*.

  Missing and *wrong* variables occur when a developer does not include the appropriate variable in a log entry or uses the wrong one, respectively. Even when the correct variable is logged, it might not have a defined string form, so logging it produces a *malformed output* that can produce unexpected, long, and confusing logs. Finally, log entries documenting similar events might *vary the granularity* by arbitrarily providing different levels of detail or including different variables.

- **Implications**: Missing and wrong variables, as well as malformed output, create confusion and render the logs less useful. Varying granularity could cause information loss or lead the developer to think similar codes work differently.

- **References**: Bijvank et al. (2013); Chen and Jiang (2017, 2019); Patel et al. (2022); Yuan et al. (2012b)

- **Example**: The example modifies the code presented in Listing 1. Line 5 uses variable `conn` as part of an error log entry, when it should log variable `e` instead. The issue is that `conn` is an object, not the name of the database or a description of the occurred error, which can lead to a malformed output if the object does not have a defined string presentation or logs irrelevant information. Another issue may occur if the logging statement on line 5 indicating a failed connection cannot be associated with the database, as it might crucial for debugging to be identify which database was involved.

```
1  try:
2      logger.info(f'Connecting to database')
3      conn = sqlite3.connect(db_file)
4  except sqlite3.Error as e:
5      logger.error(f'Failed to establish connection: {conn}. Abort proc.')
```

  An example of the *varying granularity* facet could be related to the accuracy with which a variable is logged. For demonstrating the smell, we present a piece of code related to currency exchange rate. The code logs the exchange rate if it is higher or equal than a predefined threshold. Here, the threshold for a high rate has six decimals, while the rate is logged using only three. This can cause confusion as the log suggests the comparison was done with three decimals instead of six, which could potentially lead to small exchange rates, such as 1.555111, to be logged.

```
if exchange_rate >= 1.555555 :
    logging.warning(f"High exchange rate for currency_pair: {exchange_rate:.3f}")
```

Real-life examples are presented for instance in Yuan et al. 2012b, Fig 5; Chen and Jiang 2017, Fig 4; Patel et al. 2022, Tab 14, example 7.

LS5: Variable on the edge

- **Description**: As in any other code, the variables in the logging code need to be used safely, as unsafe usage can create issues during the program execution. The facets of this smell include *uninitialized variables*, *null objects or references*, and *explicit casts*.

  Uninitialized variables are variables which are used before they have been given a value. Therefore, printing them or performing operations can result in an unpredictable outcome. *Null objects and references* are objects which do not have a value or references whose value is implicitly indicating that they do not point to a valid object; using these references/objects has similar consequences as uninitialized variables. *Explicit cast* might force a conversion to an unsuitable type.

- **Implications**: Using "variables on the edge" could lead to output some incorrect information or in the worst case to crash the program.

- **References**: Chen and Jiang (2017, 2019); Patel et al. (2022)

- **Example**: The example modifies the logging code presented in Listing 1. On line 2, the variable is used before it is defined; this can lead to printing something unknown or outdated value if a variable with the same name was previously used in the code. In both cases the log would contain wrong information. Alternatively, the logging statement could crash the program.

```
1 try:
2     logger.info(f'Connecting to database {conn}')
3     conn = sqlite3.connect(db_file)
4 except sqlite3.Error as e:
5     logger.error(f'Failed to connect to database: {e}. Abort proc.')
```

  Real-life examples are presented in Chen and Jiang 2017, Fig 4 and Patel et al. 2022, Tab 14, example 6.

LS6: Message madness

- **Description**: Logging messages are the static text of a log entry; this smell includes the issues related to them. The facets in the case of a single log entry are *missing*, *wrong*, and *imprecise* messages, as well as *unnecessary stack traces* and *language issues*. For multiple log entries, the facets are *inconsistent* and *duplicated* messages.

  Inconsistent messages occur when messages that should be similar are actually different, for example, due to differences in granularity. However, a *duplicated message* can be also an issue in some cases. *Language issues* can affect any message; they include, for instance, typos, grammar issues (including wrong verbe tense), unclear language, and incorrect translations.

- **Implications**: The logs are misleading, difficult to comprehend; locating the source of the log from the code becomes more laborious.

- **References**: Bijvank et al. (2013); Bogatinovski et al. (2022); Chen and Jiang (2019, 2022); Ding et al. (2023); Gu et al. (2023); Hassani et al. (2018); He et al. (2022); Li et al. (2021a,b, 2022, 2017b); Lu et al. (2015); Liu et al. (2020); Patel et al. (2022); Shen et al. (2023); Yuan et al. (2012b); Zhao et al. (2023).

- **Example**: The example below has several differences compared to the log shown in Figure 4. Already line 1 demonstrates multiple issues. First, it uses the wrong verbe tense, as it indicates the connection was already established, when in reality the program is still connecting to the database. Second, it is inconsistent with line 8, even though both entries record the same event. Line 2 uses unclear language, as using the word “slow” would be more clear. Finally, line 3 is missing the message about failing to establish the connection.

```
1 2024-07-29 15:09:51Z INFO [database] Connected to database
2 2024-07-29 15:11:36Z WARN [database] Connection stagnant , retrying
3 2024-07-29 15:12:03Z [database] ERROR ␣
...
8 2024-07-26 16:34:12Z INFO [database2] Connecting to database
```

  Real-life examples are provided for instance in Chen and Jiang 2019, Fig 8; Yuan et al. 2012b, Fig 9; and Patel et al. 2022, Tab 13.

LS7: Logging lost in the wind

- **Description**: Missing logging indicates that the logging code or the log file lacks relevant logging entries. The facets are *not having a logging entry in the code* and setting the *logging verbosity to a too scarce level* at runtime.

- **Implications**: Missing entries make the logs less useful as important runtime information is not recorded. This might result in some issues in the software going unnoticed. If the system logs only issues from a certain logging level and ignores others, debugging an issue might become difficult as the granularity of the recorded information is not sufficient; moreover, mistakes in assigning logging levels become more severe.

- **References**: Chen and Jiang (2022); Fu et al. (2014); Hassani et al. (2018); He et al. (2022); Li et al. (2021a, 2017b); Lu et al. (2015); Patel et al. (2022); Shen et al. (2023); Yang et al. (2021); Yuan et al. (2012a); Zhaoxue et al. (2021)

- **Example**: Compared to the example log shown in Figure 4, the below example is missing a log entry. It does not indicate whether a connection was established to the database. This is considered a missing log entry and could cause difficult debugging as there is no record of the connection failing due to a timeout.

```
1 2024-07-29 15:09:51Z INFO [database] Connecting to database
2 2024-07-29 15:11:36Z WARN [database] Connection slow, retrying
␣
...
8 2024-07-26 16:34:12Z INFO [database2] Connecting to database
```

  A real-life example is provided in Patel et al. 2022, Sec 4.3.1.

LS8: Landfill logs

- **Description**: Issues related to having too much logging. The facets are *useless*, *redundant*, and *too detailed* log entries. Note, that this smell is context dependent as 'too much' and 'useless' depend on the task the log is used for. For example, the issue could be caused by placing logging in a tight loop or not properly aggregating information.

- **Implications**: This smell hides important information in the logs and affects the efficiency of the system where the logging occurs. Excessive logging can also make the log files impossible to analyze manually and thus introduce the need for tools and automation as part of the logging system. A large amount of logs can also affect the logging file management as large files might become too big to open or join with another file and require more storage space.

- **References**: Ding et al. (2015); Fu et al. (2014); Gu et al. (2023); Hassani et al. (2018); He et al. (2022); Li et al. (2021a); Lu et al. (2015); Marron (2018); Patel et al. (2022); Zeng et al. (2019)

- **Example**: The example extends the log presented in Figure 4. At line 1, the program attempts to connect to a database. However, due to the high volume of logging, the slow connection (line 3471) and eventual failure to connect (line 7392) are reported thousands of lines apart. In addition to that, line 3470 presents a completely unnecessary log entry, while line 3472 is relevant but most likely too detailed for the majority of the use cases of the log.

```
1 2024-07-26 12:39:19Z INFO [database] Connecting to database
...
3470 2024-07-26 12:40:46Z DEBUG: The quick brown fox jumps over the lazy dog.
3471 2024-07-26 12:41:46Z WARN [database] Connection slow, retrying
3472 2024-07-26 12:41:46Z INFO Server started on port 8080 at 2024-07-26 12:41:43 | Environment: Prod
    | Server Details: Java Version: 11.0.11, Memory Allocation: 8GB RAM, CPU Cores: 4, Operating
    System: Linux (Ubuntu 20.04.3 LTS) | Additional Notes: Admin Contact: admin@example.com,
    Backup Schedule: Daily, 02:00 UTC, Log Rotation: Weekly, Sundays at 00:00 UTC
...
7392 2024-07-26 12:42:16Z ERROR [database] Failed to establish connection: Connection timeout.
    Abort proc.
```

  Marron 2018, Fig 1 provides a real-life example in JavaScript; Hassani et al. 2018, p. 3265 describes a real-life scenario when a project produced too much logging; and Patel et al. 2022, Tab 16 example 9 present a case of removing redundant information from a log entry.

LS9: Sleeping guards

- **Description**: Logging guards encapsulate a piece of logging code and control whether it is executed or not. The facets for this smell are *missing guards* and *incorrect guards*.

  Missing guards result in the code associated with a logging statement being executed regardless of whether an actual log entry is recorded. *Incorrect guards* have conditions that do not match their intended execution such as requiring a different logging level than the one used in the logging entry.

- **Implications**: Incorrect or missing sleeping guards negatively affect the performance of a program when code related to logging is executed but no log entries are created based on the execution code. This uses resources without producing anything. Additionally, having only part of the logging code inside logging guards could lead to undefined or outdated variables which might crash the program or produce an erroneous log entry.

- **References**: Chen and Jiang (2019); Zhi et al. (2022)
- **Example**: The smell is demonstrated with a piece of code retrieving the contents of an HTML page. The code is meant to create and log a specifically formulated report of the content when the logging verbosity of the program is set to DEBUG. However, without the logging guards the report is formulated every time the code is executed, regardless of the currently used logging verbosity. This could cause performance issues as the report formulation could require a lot of resources depending on the amount of data on the web page.

```
logging.debug("Starting webpage processing")
html = fetch_webpage(url)
titles = parse_webpage(html)
<missing logging guard>
    report = generate_debug_report(titles)
    report_str = "Report Summary: {report['summary']}, Titles: ..."
logging.debug("Generated report: {report_str}")
```

  Real-life example of added logging guards is provided in Chen and Jiang 2019, Fig 9 (ALG).

LS10: Skeleton in the closet (logging code smells)

- **Description**: Issues related to the code used to create the log. This covers the code required in preparation for creating the actual logging entry, such as condition checks, data aggregation, variable conversions as well as the code doing the actual logging. This smell can be considered a *meta log smell* as it includes any kind of quality issues affecting the logging code, such as *code smells* like long and duplicated code, as well as issues in *comprehensibility*.
- **Implications**: Code issues (smells) weaken the clarity and comprehensibility of the code and thus have an effect on the maintainability. However, it is worth noting that just having logging among feature code can sometimes decrease the clarity of the feature code. These issues might also affect the performance and resource usage of the code.
- **References**: Chen and Jiang (2017, 2019, 2022); Gholamian and Ward (2020); Li et al. (2021a); Zhi et al. (2022)
- **Example**: The two functions below are affected by *smells of the logging code*. More specifically, they are affected by the “duplicated code” code smell as it is unnecessary that both functions connect to a cursor. Additionally, they are subject to the “inappropriate intimacy” code smell as the logging statements are coupled with the database operations, which makes it more difficult to change the logging without affecting the database code.

```
def insert_user(conn, name, age):
    logging.debug(f"Inserting user: {name}, Age: {age}")
    cursor = conn.cursor()
    cursor.execute('''INSERT INTO users (name, age) VALUES ...)
    conn.commit()

def fetch_user(conn, user_id):
    logging.debug(f"Fetching user with ID: {user_id}")
    cursor = conn.cursor()
    cursor.execute('''SELECT * FROM users WHERE ...)
    user = cursor.fetchone()
```

```
        logging.debug(f"Fetched user: {user}")
        return user
```

A real-life example of code duplication in logging is shown by Chen and Jiang 2017, Fig 4.

5.1.3 Consequences of log smells

The third category is the consequences of log smells. As mentioned earlier, we focused only on the direct consequences of log smells. The issues, their descriptions, the log smells causing them, and their implications are presented below.

CO1: Information leak

- **Description**: Log files might be publicly accessible to anyone using a software. In such cases printing sensitive information in the log file could cause an information leak. The data includes network, database, location, and user account-sensitive data, as well as kernel pointers and cryptographic keys.
- **Implications**: The leaked information might make the software more vulnerable to attacks as the information can be used against the software. Alternatively, the leaked data might be the personal data of the people or companies using the software and could be used directly against them or simply lead to a privacy breach.
- **Causes**: “Mercurial logging level” (LS3) could cause it when certain logging levels are used for internal debugging, but then a log entry containing sensitive information is labeled with an incorrect logging level. A similar situation may happen with smell “Sleeping guards” (LS9), as the leak could be caused by poorly defined logging guards. As a sign of “Landfill logs” (LS8), developers might produce too detailed logging and, while do so, they might log —by accident— sensitive information. Finally, the negative impact on code comprehensibility of ‘Skeleton in the closet” (LS10) could make the code more prone to leaks.
- **References**: Gu et al. (2023); Li et al. (2021a, 2017b); Patel et al. (2022); Zhou et al. (2020)
- **Example**: The example code connects to a database and logs all the data in it. If the access to the resulting log file is not restricted,it could potentially leak sensitive user information.

```
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
cursor.execute("SELECT * FROM users WHERE id= ...)
user_data = cursor.fetchone()
 logging.info("Fetched data: user_data")
conn.close()
```

A real example of an information leak is provided in example 13 of Table 17 in Patel et al. (2022).

CO2: Temporal inconsistency

- **Description**: Temporal inconsistency is created when the temporal relation of events in the code and the relation inferable from the log file differ from each other. For example, a log entry might mistakenly indicate that a process was completed when it is was only started; also, due to concurrency, the order in which log entries are written in the log file could not reflect the real order of the corresponding (logged) actions. This issue is particularly problematic in a black-box situation, where the reader of a log file cannot access the source code and therefore cannot verify the order of logged events.

- **Causes**: "Message madness" (LS6) as language mistakes, particularly in verb tenses, within the message of a logging entry can create a misleading impression of the code's execution. Additionally, missing log entries, i.e., symptoms of "Logging lost in the wind" (LS7), might make linking related log entries difficult and hence cause confusion interpreting the order of events in the log. The same logic applies to smell "Undercover identifier" (LS2). The "Skeleton in the closet" (LS10) smell an affects the overall quality of the logging code and hence makes creating such inconsistencies more likely.

- **Implications**: Interpreting the log files is misleading, confusing, and more laborious.

- **References**: Ding et al. (2023)

- **Example**: The example is based on the logging code shown in Listing 1 and related to the log file shown in Figure 4. The logging code at line 2 has the wrong tense as it is executed before trying to connect to the database. Consequently, based only on the line 1 of the log file, the connection is already made. This confuses the reader as the following log entries report a slow connection and failure to connect to the database, while in reality, the connection was never established, and the system had stopped trying to connect.

  Logging code:

```
1 try:
2     logger.info('Connected to database')
3     conn = sqlite3.connect(db_file)
4 except sqlite3.Error as e:
5     logger.error('Failing to connect to database: {e}. Abort proc.')
```

  Log file:

```
1 2024-07-29 15:09:51Z INFO [database] Connected to database
2 2024-07-29 15:11:36Z WARN [database] Connection slow, retrying
3 2024-07-29 15:12:03Z [database] ERROR Failing to connect to database: Connection timeout.
   ↪ Abort proc.
```

  Additional examples of temporal inconsistencies are presented in Ding et al. (2023).

CO3: Effect on system performance

- **Description**: Running the logging code and writing the logs causes (non-constant) *performance overhead*. The execution of the code can be either expected or *parasitic* in the case that logging is disabled, but logging code is still executed without it outputting anything.

- **Causes**: The performance overhead caused by logging can become an issue when the volume of logging is high ("Landfill logs" (LS8)). In this case the cost is visible from the logs, but the parasitic cost caused for example by poorly set logging guards is more difficult to identify ("Sleeping guards" (LS9)). Smell "Skeleton in the closet" (LS10) might affect the efficiency of the code.

- **Implications**: The effects on the performance can be seen in several aspects of the system, such as running speed of CPU, memory usage, overhead in disk/IO bandwidth and storage, battery consumption, response time, and the impact these conditions have on the end-user experience. Developers may not be fully aware of the performance cost associated with logging; moreover, determining the optimal amount of logging is challenging because the workload of the system is not constant.

- **References**: Chen and Jiang (2017, 2022); Ding et al. (2015); Fu et al. (2014); Gu et al. (2023); Li et al. (2021a); Marron (2018); Zeng et al. (2019); Zhao et al. (2017a)

CO4: Accidental side-effects of logging code

- **Description**: Executing logging code can change the status of the software.
- **Causes**: This could be caused, for example by the "Variable on the edge" (LS5) smell when a variable is used carelessly and result in crashing the program or another unexpected behavior. Alternatively, the issue could be caused by the "Landfill logs" (LS8) as the unnecessary logging code makes the code less comprehensible. Also, smell "Skeleton in the closet" (LS10) has a negative effect on the code comprehensibility and thus makes it more prone to accidental side-effects.
- **Implications**: Logging code can alter the status of the program in many ways. The developers might face, for example, bugs or crashes that are difficult to locate, and this can affect both the developers as well as the users of the software.
- **References**: Marron (2018); Yuan et al. (2012b)
- **Example**: Unexpected behavior could occur in a program using several threads, as the program could be affected by deadlocks if the logging mechanism is not thread-safe. A real-life example of a misplaced logging statement corrupting the software state is presented in Yuan et al. 2012b, Fig 3.

5.2 RQ2: Tools detecting logging issues

As shown by answering RQ1, we identified several log smells. While not all of the smells can be fully addressed automatically, many of them could be detected or solved to some extent using automated tools. In this RQ, we focus on identifying the tools that *detect log smells*, i.e., tools that highlight log smells but do not propose a solution for them.

From the survey, we identified 8 tools detecting log smells and 10 tools detecting other kinds of logging issues. Since we focus on log smells, we only provide a short description of the tools detecting other logging issues in B. Below, we present the description of the tools detecting log smells, followed by Table 5 summarizing the mapping between the tools and the facets of log smells that can be detected by them:

- **CLIF** (Lal et al., 2019) is a tool detecting whether logging should be added in if-blocks. It uses a three-level machine learning based model and utilizes cross-project data.
- **DLFinder** (Li et al., 2022) detects problematic duplicated logging statements. The tool uses an abstract syntax tree, data flow, and text analysis to detect five different types of issues.
- **ECLogger** (Lal et al., 2016b) detects missing logging in catch-blocks. It is an ensemble-based tool utilizing cross-project data.
- **LCAnalyzer** (Chen and Jiang, 2017) detects from logging code nullable objects, explicit cast, wrong verbosity level, long logging code (duplication of method or local variable), and malformed output. It is a static code analyzer created with Java Development Tools.
- **LogBugFinder** (Hassani et al., 2018) detects typos, missed exception messages, missing log level guards, and incorrect log levels. All issues have their own checkers, which are combined into a single tool.
- **QuLog** (Bogatinovski et al., 2022) evaluates the correct log level and detects if the messages have sufficient linguistic structure. It utilizes deep learning and explainable AI methods.

Table 5: Mapping between facets of log smells and tools detecting them.

ID	Log smell	Facet	Tools
LS1	Format turmoil	Inconsistent, incomplete, several, no format	-
LS2	Undercover identifier	Missing, vague, wrong	-
LS3	Mercurial logging level	Incorrect	LCAnalyzer, LogBugFinder
		Inconsistent levels	Yuan2012
		Missing	-
LS4	Deceptive variable	Malformed output	LCAnalyzer
		Missing, wrong, varying granularity	-
LS5	Variable on the edge	Nullable objects, explicit cast	LCAnalyzer
		Uninitialized variables	-
LS6	Message madness	Language issues	
		* Typos	LogBugFinder
		Duplicated message	DLFinder
		Missing, wrong, imprecise, inconsistent, unnecessary stack traces	-
LS7	Logging lost in the wind	Missing logging:	
		* If-blocks	CLIF
		* Catch-blocks	ECLogger
		* Exception message	LogBugFinder
		Too scarce logging verbosity	-
LS8	Landfill logs	Useless, redundant, too detailed	-
LS9	Sleeping guards	Missing	LogBugFinder, Zhi2022
		Incorrect	-
LS10	Skeleton in the closet	Long logging code, duplicated code	LCAnalyzer
		Clarity, comprehensibility	-

- **Yuan2012** (Yuan et al., 2012b) identifies inconsistent logging levels between code clones. The tool compares code clones and checks that, if the clones have logging code, the assigned verbosity levels are consistent.
- **Zhi2022** (Zhi et al., 2022) identifies missing and partial logging guards in pre-logging statements. The pre-logging statements are detected using intra-procedual data dependence analysis, intra-procedual control dependence analysis, and inter-procedural side-effect analysis.

To understand which log smells the aforementioned tools address, we compared them with the facets described in Section 5.1.2. Table 5 lists all of them; if a tool detects a facet, it is listed in the *Tool* column. Facets not having a detection tool are marked with the value “-”. Some tools address only specific parts of a facet, and these sub-facets are indicated using an asterisk (*). The table reveals that most smells have detection tool(s), and these tools cover a wide range of facets. Only the smells “Format turmoil” (LS1), “Undercover identifier” (LS2), and “Landfill logs” (LS8) do not have any tools focusing on them. In contrast “Mercurial logging level” (LS3), “Message madness” (LS6), “Logging lost in the wind” (LS7), and “Sleeping guards” (LS9) smells have multiple tools addressing them.

The tools mostly focus on different facets of the smells, covering several facets for most smells. However, none of the smells have all of its facets covered. The most commonly undetected facets are related to “missing” or “wrong” elements in logging and half of the smells (LS1, LS2, LS3, LS4, and LS6) lack a tool to identify at least one of the aspects. These undetected issues include checking for log format usage, missing variables in log entries, and incorrect static messages.

It is worth noting that each tool might only partially cover a facet. In LS7, CLIF covers only missing logging in if-blocks, ECLogger detects them in catch blocks, and LogBugFinder focuses on exception messages. For smell LS6, the facet "Language issues" consists of several sub-facets such as grammar issues and incorrect translations. However, LogBugFinder covers only typos, which leaves most of the issues related to the facet undetected.

5.3 RQ3: Tools repairing logging issues

While the answer to RQ2 identified tools detecting log smells, by answering RQ3 we focus on tools for repairing them. Tools that detect an issue and propose a solution for it are referred to as *repair tools*. From the survey, we identified 13 repair tools focusing on log smells and 5 tools for repairing logging issues related to something other than log smells. Since we focus on log smells, we only provide a short description of the tools repairing other logging issues in B. Below, we describe the 13 repair tools focusing on log smells:

- **DeepLV** (Li et al., 2021c) makes suggestions for logging levels; it is a deep-learning-based method using the syntactic context and message features.
- **Errlog** (Yuan et al., 2012a) proactively inserts logging statements into source code, particularly for unlogged exceptions. It uses the Saturn framework (Aiken et al., 2007) to implement a novel algorithm that identifies potential expectation locations and checks if they are already handeled, all while minimizing performance overhead.
- **He2018** (He et al., 2018) adds logging descriptions based on existing ones; it is based on information retrieval.
- **Kim2020** (Kim et al., 2020) assesses the appropriateness of log levels and recommends alternative levels if needed. It uses machine learning methods that learn the semantic and syntactic features of log messages.
- **LACC** (Gholamian and Ward, 2020) predicts the location and severity of missing log statements in code clones. It detects methods with logging code, calculates log-aware source code features for them, and finally makes predictions using machine learning.
- **Li2017** (Li et al., 2017a) suggests a log level for new logging statements. It uses ordinal regression models to learn the levels from project history.
- **Li2021** (Li et al., 2021b) suggests logging for exception stack traces. Predictions are made using machine learning models that are trained with several code metrics.
- **Liu2020** (Liu et al., 2020) automatically generates descriptive texts for logging statements. It reduced the problem to a retrieval-based Q&A task.
- **Liu2021** (Liu et al., 2021) recommends which variables to log and is able to consider also out-of-vocabulary words. The tool addresses this as a representation problem and trains a neural network for presenting program tokens and based on that predicts whether an identifier should be logged.
- **LTID** (Zhao et al., 2023) propagates missing IDs to logs. The method uses log graphs and ID variables in log templates.
- **PADLA** (Mizouchi et al., 2019) dynamically adjusts the logging verbosity level of a running system for better recording of irregular events. It is an extension of Log4j logging framework employing an online phase detection algorithm.

Table 6: Mapping between facets of log smells and tools repairing them.

ID	Log smell	Facets	Tool
LS1	Format turmoil	Inconsistent, incomplete, several, no format	-
LS2	Undercover identifier	Missing	LTID
		Vague, wrong	-
LS3	Mercurial logging level	Incorrect	DeepLV, Kim2020, Li2017, LACC, QuLog
		Inconsistent	VerbosityLevelDirector
		Missing	-
LS4	Deceptive variable	Missing	Liu2021
		Malformed output, wrong, varying granularity	-
LS5	On the edge variable	Nullable objects, explicit cast, uninitialized variables	-
LS6	Message madness	Missing / inconsistent (adds descriptions)	He2018, Liu2020
		Stack trace usage	Li2021
		Wrong, imprecise, duplicated language issues, sufficient linguistic structure	-
LS7	Logging lost in the wind	Missing	
		* Exceptions	Errlog
		Too scarce logging verbosity	PADLA
LS8	Landfill logs	Too detailed	PADLA
		Useless, redundant	-
LS9	Sleeping guards	Missing, incorrect	-
LS10	Skeleton in the closet	Clarity, comprehensibility, long and duplicated code	-

- **QuLog** (Bogatinovski et al., 2022) evaluates the correct log level and detects if the messages have sufficient linguistic structure. It utilizes deep learning and explainable AI methods.
- **VerbosityLevelDirector** (Anu et al., 2019) provides guidance on certain type of problematic verbosity levels. The tool does detects exception handling and condition check code blocks, produces features for them and creates a prediction based on them. The features are related to triggered methods, logging content, exception type, post process, and code comment.

Similar to the results presented for detection tools, Table 6 links the repair tools to the different facets of the log smells. The table indicates that repair tools also target most of the log smells. Only smells Format turmoil (LS1), Sleeping guards (LS9), and Skeleton in the closet (LS10) do not have tools focused on them. The most addressed smell is the Mercurial logging level (LS3), as 6 out of 13 repair tools focus on that smell.

In two cases, there is an overlap in terms of the target facets of repair tools, for example, incorrect logging level (LS3) and missing or inconsistent logging message (LS6) both have several tools addressing them. Despite targeting the same facets, these tools use different approaches and, therefore, they might detect different parts as problematic and further provide different solutions to them. The repair tools do not focus on code-related issues, but are targeted more for mistakes in the generated log entry. Also with repair tools, a tool might cover only a part of a facet; for example, Errlog addresses missing logging only in exceptions.

6 Discussion

The concept of "smell" is widely adopted in software engineering to describe practices that may be potentially harmful. Understanding them can help developers produce higher quality products, and therefore, researchers have created taxonomies for different types of smells. Although several studies discuss logging issues, we identified only two previous studies that have used the term "log smell". Both of these studies focused on specific smells rather than providing a general definition or a comprehensive taxonomy.

In this paper, we have created a taxonomy of 10 log smells and identified five causes and four consequences associated with them. Each smell corresponds to a different aspect of logging and has multiple facets that can manifest the smells. The need for addressing logging issues has long been recognized in the research community, and the topic has been extensively researched. Therefore, we have examined whether the logging tools introduced by the research community address the identified smells. We have identified eight tools designed to detect log smell and 13 tools for repairing them.

The log smells, their facets, and the tools addressing them are summarized in Table 7. The survey comprehensively investigated logging-related issues, revealing that *logging can be affected by a large variety of smells at all stages of the logging process*. Regardless of the smell, if the emerging issues are not taken seriously, they might create or evolve into other problems. Therefore, developers should be aware of the smells and their effects on the quality of logging. However, some of the smells can have a more widespread negative impact than others. For example, "Landfill logs" (LS8) can affect performance, storage costs, and comprehensibility of the logs; this smell was considered a potential cause for all four identified consequences of log smells. On the other hand, "Message madness" (LS6) mainly affects the usefulness and comprehensibility of the log entries and was identified as potential cause only once. Three smells (LS1, LS4, LS7) were not determined to be a cause for any of the consequences. However, this does not imply that these are not potentially harmful. The paper focused on direct consequences, and while none of these smells might directly cause a significant problem, they all hinder the comprehensiveness, completeness, and thus usefulness of the log, making the downstream tasks more difficult.

The results show that nine out of the smells are covered by tools, but closer inspection reveals that both detection and repair tools only cover 10 out of 32 facets. This suggests that *log smells are moderately covered by tools, but further research is needed.* The low ratio of covered facets might partly be explained by some of the log smells having solutions that are not highlighted by the survey, either because it did not identify all tools or because a relevant tool was developed for some other area of software engineering rather than logging. A project could have adopted such tools for other reasons besides logging, but as a side effect, they also improve the quality of logging. This might be true, especially for the facets that were found to have only detection tools. For example, several tools have been developed for detecting and fixing code smells (Lacerda et al., 2020), and therefore, the "Skeleton in the closet" (LS10) log smell, i.e., issues related to the *code* used to create the log, has solutions which are not reflected in this study.

Tables 5 and 6 highlight that *the focus of the identified tools varies*. For detection tools, only incorrect logging levels, missing logging guards, and missing logging are addressed by several tools, while for repair tools, incorrect logging levels and missing messages are covered multiple times. Additionally, while some of the tools cover several smells, most focus on a single one. Notably, almost all repair tools focus only on one smell, the exception being LACC, which predicts the location and severity level of logging statements. Comparing the detection and repair tools (Table 7) shows that only three facets have a detection and a repair tool, and thus, they mostly focus on different facets.

Only smell "Format turmoil" (LS1) had neither detection nor repair tools addressing it. This was unexpected, as one of the main tasks for such a tool would be parsing the log files, and several such parsers already exist (Zhang et al., 2023). However, a complete lack of solutions for LS1 might not be true, as logging libraries are commonly used, and they automatically format the logs according to specification. Additionally, we excluded from the study solutions that proposed a new approach to logging instead of focusing on solving

Table 7: Tool support for detecting (DT) or repairing (RT) the facets of the different log smells

ID	Log smell	Facets	DT	RT
LS1	Format turmoil	Inconsistent, incomplete, missing, several	-	-
LS2	Undercover identifier	Missing	-	✓
		Vague, wrong	-	-
LS3	Mercurial logging level	Incorrect	✓	✓
		Inconsistent	✓	✓
		Missing	-	-
LS4	Deceptive variable	Missing	-	✓
		Malformed output	✓	-
		Wrong, varying granularity	-	-
LS5	Variable on the edge	Nullable objects, explicit cast	✓	-
		Uninitialized variables	-	-
LS6	Message madness	Missing, inconsistent	-	✓
		Stack trace usage	-	✓
		Duplicated message	✓	-
		Language issues	✓	-
		Wrong, imprecise	-	-
LS7	Logging in the wind	Missing	✓	✓
		Too scarce logging verbosity	-	✓
LS8	Landfill logs	Too detailed	-	✓
		Useless, redundant	-	-
LS9	Sleeping guards	Missing	✓	-
		Incorrect	-	-
LS10	Skeleton in the closet	Long logging code, duplicated code	✓	-

a particular issue. These systems, such as Log++ (Marron, 2018), often included the element of uniform logging (see B).

The main facets lacking automation are logging wrong or vague data, as identifiers, variables, and messages all lack such tools. However, identifying wrong or vague information is not a trivial task and sometimes would require domain knowledge of the project. Another area that needs more focus is the quantity of logging. The tools addressing "Logging in the wind" (LS7) focus on specific cases of missing logging. Therefore, there is a need for creating more general tools or combining the existing tools into one detecting more than just one specific case. As stated previously, "Landfill logs" (LS6) could have a significant impact on the project. However, we did not identify tools for identifying useless or redundant logging. The usefulness of a log is dependent on the downstream task, and therefore, several tools would be needed to address the facet properly.

Our findings are in line with recent studies on logging. In their systematic mapping study, Gu et al. (2023) concluded that one of the aspects requiring more research focus is how "good" is the logging, i.e., the degree of how well the intentions and concerns of logging can be implemented. They also noticed that most tools are focused on the "where and what to log" categories. Our results are similar as we identified several tools addressing the where-to-log issues; the identified log smells indicate that many aspects of "how good logging is" still require more attention. In an interview conducted at Microsoft by He et al. (2022), the developers voiced a need for off-the-shelf logging tools that help the developers. This is also visible in our results, as even though several tools exist, many address only one logging issue. This makes adopting them more difficult as the developers would need to use several tools to address all relevant logging issues.

We acknowledge that we have defined smells at a higher level than in some other areas of software engineering. For instance, in our taxonomy duplicated logging code is categorized under the log smell "Skeleton in the closet", which contains general logging code-related issues. This smell includes code smells, with "Duplicated code" being a a code smell on its own. However, we found that using a higher-level groupings and then providing the facets for each smell is more practical and easier to comprehend than listing 32 separate smells.

7 Threats to Validity

This section discusses the validity threats of the study. We have divided the section according to Yin (2014) into construct validity, internal validity, external validity, and reliability.

Internal validity is related to the factors affecting the data and the inferences made on it. The conducted survey used Google Scholar as its only data source. We did not snowball the results, and we did not follow the systematic literature review methodology. Therefore, it is likely that not all relevant papers were identified during the process. The threat was mitigated by using several queries with different words describing logging issues. We also followed the SLR reporting guidelines when applicable and provided all data in the replication package.

The threat is even higher for the tool data. First, the search queries did not include any words related to tools. This is not necessarily a threat, but we might have missed some relevant tool papers for not having a query specifically for identifying tools. However, we mitigated this threat by including all tools reported in literature review papers included in the survey. Those reviews included several tools our survey had not identified. Second, several facets of the log smells could be detected or repaired with tools created for other aspects of software engineering. For example, several tools exist for identifying and fixing variable-related issues and code smells. The survey focused only on logging, and we did not explore tools not indicated in our survey.

Due to resource limitations, the data extraction was conducted by a single author without verification by another author. Therefore, the data might contain mistakes like misinterpreting text or missing relevant information. To mitigate the threat, the author responsible for conducting the survey reviewed all extractions after the first extraction round. Additionally, all authors discussed and solved issues related to the survey methodology, included papers, and any other problematic aspects arising during the extraction phase. Although it does not entirely remove the threat, as the goal of data extraction was to identify issues without making interpretations, we do not consider this a major threat.

In addition to the data extraction, the results could be to some extent subjective. Different persons could have classified the identified issues differently which would created differences to the taxonomy and the identified tools. For example, tools addressing the "where to log" problem were excluded from the results, but they could have been labeled to address smell "Logging in the wind" (LS6). Similarly, we excluded tools reporting a whole logging system though they could have also been considered as tools simultaneously addressing several issues. We mitigated this threat by conducting a card sort with all the authors, comparing the results and iteratively correcting the classification and discussing it until we reached a consensus.

A common threat affecting literature surveys is *publication bias*, i.e., positive results are more likely to be published. The selection criteria used in the survey did not require the included papers to provide a solution to the reported logging issue, and therefore, publication bias is not considered a major threat in this survey. The selection criteria were also selected for obtaining a representative view of the current issues, so the criteria did not pose strict exclusion criteria based on paper type or publication venue.

Construct validity is related to how well the adopted measures represent the theoretical constructs investigated in the study. The concept of smell is already well-established in other areas of software engineering. To keep the term consistent, we formulated the definition of a log smell based on the existing definitions of smells in other areas and also the definitions provided in the context of logging.

External validity is concerned with the generalizability of the results outside the scope of the study. In a survey the generalizability depends on the included primary studies. In this study, the goal was to collect as comprehensive a list of logging issues as possible; therefore, the varying quality and scope of the included studies were not considered a threat. However, the survey excluded gray literature, which could be a relevant source for identifying issues the developers are facing while logging. Some issues might be known only in the industry, and no research has yet been published on those issues. However, given the large variety of issues identified in the conducted survey, it is reasonable to believe that the created taxonomy covers the main issues affecting logging and, therefore, to some extent, generalizes also to the industrial context.

Reliability is related to the trustworthiness and repeatability of the obtained results. To mitigate this threat, we documented the methodological steps and provided a replication package containing full query results, assigned labels, as well as results from conducted categorization and card sorts.

8 Related Work

8.1 Secondary studies on logging and logging issues

Logging issues have been investigated by several recent works, with the most similar to our work being the systematic mapping study by Gu et al. (2023). They investigated 56 papers to identify major issues in logging, their solutions, as well as gaps in research. The findings were categorized into *where to log, what to log, why to log*, and *how well is the logging*. The study identified eight main logging issues: “lacking crucial messages”, “redundant or useless messages”, “incorrect or ambiguous messages”, “heterogeneity of messages”, “leakage of sensitive data” related to the what to log category, “performance overhead” in the where and what to log, “maintenance barriers”, and “difficulties in validation and verification of log statements” in the how well is the logging. However, the study found no issues related to the why-to-log category; similarly, most of the solutions in their study are related to where- or what-to-log issues, which indicated gaps in the why and how well the logging areas are. However, the authors noted that the research area is active, suggesting that logging issues are not yet completely solved. The main difference to our work is the perspective from which logging issues are inspected. Our work focuses on logging issues and specifically log smells, while this work focuses on the where, what, why and how to log aspects, which is more focused on the process.

A systematic literature review by Batoun et al. (2024) inspected the state-of-the-art practices with a focus on instrumentation, storing, and preprocessing of the log data. The study investigated both scientific literature as well as practitioner Q&A forum (StackOverflow). The authors identified a gap between the issues practitioners face and the scientific literature as five out of seven high-level topics identified from StackOverflow were not covered by the scientific literature, such as infrastructure-related configuration and context-dependent usage of logs. Based on the results, the authors make several suggestions for researchers and practitioners, such as ideas for future studies and practitioners adopting the solutions proposed in the scientific literature.

A survey including 112 papers from Gholamian and Ward (2022) investigates the current state-of-the-art related to the logging field. The paper investigates which areas have been studied and provides a taxonomy of the logging research. Similar to our work, the authors assess the costs, i.e., issues, of logging. While the work includes several issues such as performance cost, verbosity level issues, and noisy log files, this was not the main aspect of the study. Finally, the work presents several aspects of logging that require more research, such as cost-aware logging, log maintenance, and improved logging practices.

Zhaoxue et al. (2021) conducted a literature survey where they investigated log enhancement, log parsing, and log analysis in the context of AIOps and big data. The similarity between our work and theirs lies in the log enhancement aspect, which their work focuses on determining what and where to log. They highlight similar problems as we do and, additionally, summarize the trends and open issues in log enhancement. They summarized the log enhancing methods to use more and more fine-grained features, coupling of subsequent processes, and automation to avoid inconsistent logging.

Chen and Jiang (2022) conducted a survey of 69 papers about the log instrumentation, i.e., inserting logging code. The work focuses on logging approaches, logging utility integration, and logging code composition. The work identified nine challenges, which were classified into usability, diagnosability, logging code quality, and security compliance-related issues. In addition to the issues, the work also discusses the solutions.

He et al. (2021) provide a comprehensive overview of the current status of automated log analysis in terms of logging, log compression, log parsing, log mining, open-source toolkits and datasets, and best current practices. Their goal was to help practitioners understand the usage of these techniques and the benefits and challenges associated with them. The paper has a section specifically discussing challenges related to logging; the issues are categorized using the where, what and how to log aspects. Additionally, this article mentions works detecting and evaluating anti-patterns. However, the main focus of the paper is on log analysis instead of the issues related to logging.

Cândido et al. (2021) performed a systematic mapping study to investigate the researched challenges in the life-cycle of log data. The results were categorized and reported using three main areas: logging, log infrastructure, and log analysis. The authors identified 24 papers related to logging, concluding that developers need better tools for logging, but the ambiguity of requirements for logs poses a challenge for tool creation.

8.2 Software Smells

The term "smell" in the domain of software engineering was popularized by Beck and Fowler when they characterized code smells (Fowler et al., 1999); since then, it has been widely adopted within the software engineering field. In their survey, Sharma and Spinellis (2018) identified the following 14 groups of smells:

- **Architecture** (Medvidovic et al., 2009; Brown et al., 1998): architectural decision which has a negative impact on system quality. These include, for example, "Connector envy" in which components that should be connected by a connector are not.
- **Aspect-oriented systems** (Macia Bertran et al., 2011; Alves et al., 2014): Recurring mistakes and pitfalls in aspect-oriented programming, such as "incorrect implementation logic" and "non-dedicated implementation elements".
- **Configuration systems** (Sharma et al., 2016): 13 implementation and 11 design configuration characteristics violating best practices and negatively affecting quality. These include "incomplete tasks" and "insufficient modularization".
- **Database** (Karwin, 2022) Antipatterns in SQL code. These can be divided into logical database design, physical database design, query, and application development antipatterns.
- **Design** (Binkley et al., 2008; Suryanarayana et al., 2014): A violation of a design principle or a dependence structure indicating potential problems for on-going software evolution and maintenance. These include missing abstraction and large depedence cluster.
- **Energy** (Vetro et al., 2013): Five code patterns that make running software less energy efficient. These include repeated conditionals and parameter by value.
- **Implementation** (Fowler et al., 1999; Guerrouj et al., 2017; Arnaoudova et al., 2013): The group contains several different types of smells such as code, lexical, and linguistic. Code smells are described in more detail later; linguistic smells refer to bad practices related to names and documentation of different parts of a program.
- **Models** (El-Attar and Miller, 2010; Das and Dingel, 2018): Includes issues creating quality problems in use case models, for example "multiple actors associated with one use case", as well issues in model-based development of real-time embedded software systems such as "Misuse of local transitions".
- **Performance** (Smith and Williams, 2000; Sharma and Anwer, 2014): Design antipatterns which have a negative impact on the system performance. These include "Falling dominoes" in which one failure causes further performance failures and "Traffic jam" where a problem causes a backlog of jobs and increases their response time.

- **Reuse** (Long, 2001): Antipatterns related to systematic reusing of software. They list 11 antipatterns such as the "Used car fiasco" in which software from another group is considered reusable, but the quality of the code is not high enough and its creators do not provide sufficient support.
- **Services** (Kral and Zemlicka, 2007; Palma and Mohay, 2015): An implementation hindering the maintenance and evolution of a service in a service-based system. These include "Chatty web service" and "Redundant port-types".
- **Test** (Van Deursen et al., 2001; Garousi and Küçük, 2018): Symptoms indicating potential design problems in the test code. For example: "Lazy Test", "Resource Optimism", and "Test Code Duplication".
- **Usability** (de Oliveira T. Souza et al., 2021): Issues that negatively affect the users performing any task with a system. These include misleading links, laborious tasks, and unresponsive elements.
- **Web** (Nguyen et al., 2012): Embedded code smells in server-side code such as mixing code and presentation logic or style, duplicating JS code to the server-side code, and HTML syntactic errors.

When Sharma and Spinellis (2018) investigated the definitions of different smells, they concluded that the term is usually defined as an indicator of deeper design problems, a poor solution, a recurring problem, an issue impacting quality, or something violating best practices. Hence, smells by definition do not prevent software from working as intended, but they have a negative impact on the quality of a project, and they could result in more serious problems in the long run. Even though the terms smell and anti-pattern are often used as synonyms, they have different definitions. Anti-patterns are defined as "just like a pattern, except that instead of a solution it gives something that looks superficially like a solution, but isn't one" (Koenig, 1998). Therefore, anti-patterns are conscious choices leading to negative consequences, while code smells indicate problems and are not necessarily created on purpose (Sharma and Spinellis, 2018).

8.2.1 Code Smells

The most relevant type of smell for this paper is the code smell. It is a code structure usually related to production code that indicates a potentially deeper problem and often needs refactoring. Jerzyk and Madeyski (2023) have listed 56 smells such as Duplicate Code, Feature Envy, and Dead Code. The descriptions of the smells are generally accepted within research, but as the definitions are general in nature, their detailed definitions vary between studies.

Since their definition, code smells have been subject to a significant amount of research. Sobrinho et al. (2021) reviewed 351 papers about code smells and investigated the current research using the what, where, when, which, who aspects. They found evidence that smells are linked to some negative aspects, but further research is still necessary as, for example, the co-occurrence of smells is not widely studied.

Santos et al. (2018) conducted a systematic review to investigate the impact of code smells on the software development. They categorized the research on the topic into three types of papers: correlational, role of human, and tool assessment. They investigated 64 papers and concluded that code smells do not seem to have a strong correlation with important development attributes and that smell detection done by humans should not be trusted.

Several tools have been created for detecting and refactoring code smells from code. In their literature review Lacerda et al. (2020) identified 162 distinct tools for detecting code smells and another 24 tools for refactoring them. With such high number of tools, the tools also depict very different characteristics. They differ in, for example, whether they are open-source, the degree of automation, and supported languages.

9 Conclusion

In this work, we have reported on a literature survey that has led to the definition of a taxonomy of *log smells*, containing ten log smells and their facets, as well as to the identification of five direct causes of log smells and four consequences for them. We have also defined a mapping of log smells (and their facets) to tools detecting and solving them, identifying eight tools that detect different facets from seven log smells and 13 tools that address (and repair) facets from six log smells.

The results of our study indicate that logging is prone to issues and smells at all phases, affecting both the logging code and the log files. As log smells represent logging issues that hinder the quality of logging, potentially leading to more serious consequences, it is crucial for both developers and researchers to understand, be aware of, and have methods for mitigating them. Therefore, we believe the presented overview of log smells, their causes, consequences, manifestations, and automated mitigation strategies is relevant for both industry and academia.

As part of future work, we plan to investigate selected log smells, by characterizing them with more details and empirically assess their impact in the field. We also plan to develop detection and repair tools for the log smells that lack automated solutions to handle them.

Acknowledgements

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant reference C22/IS/17373407/LOGODOR. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the authors have applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

References

A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 43–48, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935953. doi: 10.1145/1251535.1251543. URL https://doi.org/10.1145/1251535.1251543.

P. Alves, E. Figueiredo, and F. Ferrari. Avoiding code pitfalls in aspect-oriented programming. In F. M. Quintão Pereira, editor, *Programming Languages*, pages 31–46, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11863-5.

H. Anu, J. Chen, W. Shi, J. Hou, B. Liang, and B. Qin. An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 125–134, Cleveland, OH, USA, Sept. 2019. IEEE. ISBN 978-1-72813-094-1. doi: 10.1109/ICSME.2019.00022. URL https://ieeexplore.ieee.org/document/8919094/.

V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196, 2013. doi: 10.1109/CSMR.2013.28.

M. A. Batoun, M. Sayagh, R. Aghili, A. Ouni, and H. Li. A literature review and existing challenges on software logging practices: From the creation to the analysis of software logs. *Empirical Software Engineering*, 29(4):103, July 2024. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-024-10452-w. URL https://link.springer.com/10.1007/s10664-024-10452-w.

R. Bijvank, W. Wiersema, and C. Köppe. Software architecture patterns for system administration support. In *Proceedings of the 20th Conference on Pattern Languages of Programs*, PLoP '13, pages 1–14, USA, 2013. The Hillside Group. ISBN 9781941652008.

D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener. Dependence anti patterns. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 25–34, 2008. doi: 10.1109/ASEW.2008.4686318.

J. Bogatinovski and O. Kao. Auto-Logging: AI-centred Logging Instrumentation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 95–100, Melbourne, Australia, May 2023. IEEE. ISBN 9798350300390. doi: 10.1109/ICSE-NIER58687.2023.00023. URL `https://ieeexplore.ieee.org/document/10173904/`.

J. Bogatinovski, S. Nedelkoski, A. Acker, J. Cardoso, and O. Kao. QuLog: data-driven approach for log instruction quality assessment. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 275–286, Virtual Event, May 2022. ACM. ISBN 978-1-4503-9298-3. doi: 10.1145/3524610.3527906. URL `https://dl.acm.org/doi/10.1145/3524610.3527906`.

W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., USA, 1st edition, 1998. ISBN 0471197130.

B. Chen and Z. M. Jiang. Characterizing and Detecting Anti-Patterns in the Logging Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 71–81, Buenos Aires, May 2017. IEEE. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.15. URL `http://ieeexplore.ieee.org/document/7985651/`.

B. Chen and Z. M. Jiang. Extracting and studying the Logging-Code-Issue- Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering*, 24(4):2285–2322, Aug. 2019. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-019-09690-0. URL `http://link.springer.com/10.1007/s10664-019-09690-0`.

B. Chen and Z. M. J. Jiang. A Survey of Software Log Instrumentation. *ACM Computing Surveys*, 54(4): 1–34, May 2022. ISSN 0360-0300, 1557-7341. doi: 10.1145/3448976. URL `https://dl.acm.org/doi/10.1145/3448976`.

M. Cinque, D. Cotroneo, and A. Pecchia. A Logging Approach for Effective Dependability Evaluation of Complex Systems. In *2009 Second International Conference on Dependability*, pages 105–110, Athens/Glyfada, Greece, June 2009. IEEE. ISBN 978-0-7695-3666-8. doi: 10.1109/DEPEND.2009.22. URL `http://ieeexplore.ieee.org/document/5211082/`.

M. Cinque, D. Cotroneo, and A. Pecchia. Event Logs for the Analysis of Software Failures: A Rule-Based Approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, June 2013. ISSN 0098-5589, 1939-3520. doi: 10.1109/TSE.2012.67. URL `http://ieeexplore.ieee.org/document/6320555/`.

J. Cândido, M. Aniche, and A. Van Deursen. Log-based software monitoring: a systematic mapping study. *PeerJ Computer Science*, 7:e489, May 2021. ISSN 2376-5992. doi: 10.7717/peerj-cs.489. URL `https://peerj.com/articles/cs-489`.

T. K. Das and J. Dingel. Model development guidelines for UML-RT: conventions, patterns and antipatterns. *Software & Systems Modeling*, 17(3):717–752, July 2018. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-016-0549-6. URL `http://link.springer.com/10.1007/s10270-016-0549-6`.

J. de Oliveira T. Souza, A. D. de Souza, L. G. Vasconcelos, and L. A. Baldochi. Usability smells: A systematic review. In S. Latifi, editor, *ITNG 2021 18th International Conference on Information Technology-New Generations*, pages 281–288, Cham, 2021. Springer International Publishing. ISBN 978-3-030-70416-2.

R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A Cost-Aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 139–150, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL `https://www.usenix.org/conference/atc15/technical-session/presentation/ding`.

Z. Ding, Y. Tang, Y. Li, H. Li, and W. Shang. On the Temporal Relations between Logging and Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 843–854, Melbourne, Australia, May 2023. IEEE. ISBN 978-1-66545-701-9. doi: 10.1109/ICSE48619.2023.00079. URL `https://ieeexplore.ieee.org/document/10172750/`.

M. El-Attar and J. Miller. Improving the quality of use case models using antipatterns. *Software & Systems Modeling*, 9(2):141–160, Apr. 2010. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-009-0112-9. URL `http://link.springer.com/10.1007/s10270-009-0112-9`.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring : : Improving the Design of Existing Code*. Object Technology Ser. Addison Wesley Professional, Boston, 1999. ISBN 0201485672.

Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, Hyderabad India, May 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591175. URL `https://dl.acm.org/doi/10.1145/2591062.2591175`.

V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, Apr. 2018. ISSN 01641212. doi: 10.1016/j.jss.2017.12.013. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121217303060`.

S. Gholamian and P. A. S. Ward. Logging statements' prediction based on source code clones. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 82–91, Brno Czech Republic, Mar. 2020. ACM. ISBN 978-1-4503-6866-7. doi: 10.1145/3341105.3373845. URL `https://dl.acm.org/doi/10.1145/3341105.3373845`.

S. Gholamian and P. A. S. Ward. A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis, Jan. 2022. URL `http://arxiv.org/abs/2110.12489`. arXiv:2110.12489 [cs].

S. Gu, G. Rong, H. Zhang, and H. Shen. Logging Practices in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering*, 49(2):902–923, Feb. 2023. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2022.3166924. URL `https://ieeexplore.ieee.org/document/9756253/`.

L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B. C. Fung, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc. Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. *Software Quality Journal*, 25:641–670, 2017.

M. Hassani, W. Shang, E. Shihab, and N. Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 23(6):3248–3280, Dec. 2018. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-018-9603-z. URL `http://link.springer.com/10.1007/s10664-018-9603-z`.

P. He, Z. Chen, S. He, and M. R. Lyu. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 178–189, Montpellier France, Sept. 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238193. URL `https://dl.acm.org/doi/10.1145/3238147.3238193`.

S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu. A survey on automated log analysis for reliability engineering. *ACM Comput. Surv.*, 54(6), July 2021. ISSN 0360-0300. doi: 10.1145/3460345. URL `https://doi.org/10.1145/3460345`.

S. He, X. Zhang, P. He, Y. Xu, L. Li, Y. Kang, M. Ma, Y. Wei, Y. Dang, S. Rajmohan, and Q. Lin. An empirical study of log analysis at Microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1465–1476, Singapore Singapore, Nov. 2022. ACM. ISBN 978-1-4503-9413-0. doi: 10.1145/3540250.3558963. URL `https://dl.acm.org/doi/10.1145/3540250.3558963`.

M. Jerzyk and L. Madeyski. *Code Smells: A Comprehensive Online Catalog and Taxonomy*, pages 543–576. Springer Nature Switzerland, Cham, 2023. ISBN 978-3-031-25695-0. doi: 10.1007/978-3-031-25695-0_24. URL `https://doi.org/10.1007/978-3-031-25695-0_24`.

T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang, and Y. Liu. Machine Deserves Better Logging: A Log Enhancement Approach for Automatic Fault Diagnosis. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 106–111, Memphis, TN, Oct. 2018. IEEE. ISBN 978-1-5386-9443-5. doi: 10.1109/ISSREW.2018.00-22. URL `https://ieeexplore.ieee.org/document/8539172/`.

S. Karumanchi and A. C. Squicciarini. In the wild: a large scale study of web services vulnerabilities. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1239–1246, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450324694. doi: 10.1145/2554850.2555010. URL `https://doi-org.proxy.bnl.lu/10.1145/2554850.2555010`.

B. Karwin. *SQL Antipatterns, Volume 1*. Pragmatic Bookshelf, 2022.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

T. Kim, S. Kim, S. Park, and Y. Park. Automatic recommendation to appropriate log levels. *Software: Practice and Experience*, 50(3):189–209, 2020. doi: https://doi.org/10.1002/spe.2771. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2771`.

B. Kitchenham, L. Madeyski, and D. Budgen. SEGRESS: Software Engineering Guidelines for REporting Secondary Studies. *IEEE Transactions on Software Engineering*, 49(3):1273–1298, Mar. 2023. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2022.3174092. URL `https://ieeexplore.ieee.org/document/9772383/`.

A. Koenig. Patterns and antipatterns. *J. Object Oriented Program.*, 8:46–48, 1998. URL `https://api.semanticscholar.org/CorpusID:2241627`.

J. Kral and M. Zemlicka. The most important service-oriented antipatterns. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 29–29, 2007. doi: 10.1109/ICSEA.2007.74.

T. Kubota, N. Aota, and K. Kono. Logging Inter-Thread Data Dependencies in Linux Kernel. *IEICE Transactions on Information and Systems*, E103.D(7):1633–1646, July 2020. ISSN 0916-8532, 1745-1361. doi: 10.1587/transinf.2019EDP7255. URL `https://www.jstage.jst.go.jp/article/transinf/E103.D/7/E103.D_2019EDP7255/_article`.

G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, Sept. 2020. ISSN 01641212. doi: 10.1016/j.jss.2020.110610. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121220300881`.

S. Lal, N. Sardana, and A. Sureka. Improving logging prediction on imbalanced datasets: A case study on open source java projects. *International Journal of Open Source Software and Processes (IJOSSP)*, 7(2): 43–71, 2016a.

S. Lal, N. Sardana, and A. Sureka. ECLogger: Cross-Project Catch-Block Logging Prediction Using Ensemble of Classifiers. *e-Informatica Vol. XI*, page 2017; ISSN 18977979, 2016b. doi: 10.5277/E-INF170101. URL `http://www.e-informatyka.pl/attach/e-Informatica_-_Volume_11/eInformatica2017Art1.pdf`. Medium: PDF Publisher: Institute of Applied Informatics, Wrocław University of Technology, Wrocław.

S. Lal, N. Sardana, and A. Sureka. LogOptPlus: Learning to Optimize Logging in Catch and If Programming Constructs. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, pages 215–220, Atlanta, GA, USA, June 2016c. IEEE. ISBN 978-1-4673-8845-0. doi: 10.1109/COMPSAC.2016.149. URL `http://ieeexplore.ieee.org/document/7552011/`.

S. Lal, N. Sardana, and A. Sureka. Three-level learning for improving cross-project logging prediction for if-blocks. *Journal of King Saud University - Computer and Information Sciences*, 31(4):481–496, Oct. 2019. ISSN 13191578. doi: 10.1016/j.jksuci.2017.07.006. URL `https://linkinghub.elsevier.com/retrieve/pii/S1319157817300307`.

H. Li, W. Shang, and A. E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, Aug. 2017a. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-016-9456-2. URL `http://link.springer.com/10.1007/s10664-016-9456-2`.

H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865, Aug. 2017b. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-016-9467-z. URL `http://link.springer.com/10.1007/s10664-016-9467-z`.

H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. A Qualitative Study of the Benefits and Costs of Logging From Developers' Perspectives. *IEEE Transactions on Software Engineering*, 47(12):2858–2873, Dec. 2021a. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2020.2970422. URL `https://ieeexplore.ieee.org/document/8976297/`.

H. Li, H. Zhang, S. Wang, and A. E. Hassan. Studying the Practices of Logging Exception Stack Traces in Open-Source Software Projects. *IEEE Transactions on Software Engineering*, pages 1–1, 2021b. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2021.3129688. URL `https://ieeexplore.ieee.org/document/9623518/`.

S. Li, X. Niu, Z. Jia, X. Liao, J. Wang, and T. Li. Guiding log revisions by learning from software evolution history. *Empirical Software Engineering*, 25(3):2302–2340, May 2020a. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-019-09757-y. URL `http://link.springer.com/10.1007/s10664-019-09757-y`.

Z. Li, T.-H. Chen, J. Yang, and W. Shang. DLFinder: Characterizing and Detecting Duplicate Logging Code Smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 152–163, Montreal, QC, Canada, May 2019. IEEE. ISBN 978-1-72810-869-8. doi: 10.1109/ICSE.2019.00032. URL https://ieeexplore.ieee.org/document/8811945/.

Z. Li, T.-H. P. Chen, and W. Shang. 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*, pages 361–372, Virtual Event Australia, Dec. 2020b. ACM. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416636. URL https://dl.acm.org/doi/10.1145/3324884.3416636.

Z. Li, H. Li, T.-H. Chen, and W. Shang. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1461–1472, Madrid, ES, May 2021c. IEEE. ISBN 978-1-66540-296-5. doi: 10.1109/ICSE43902.2021.00131. URL https://ieeexplore.ieee.org/document/9402068/.

Z. Li, T.-H. Chen, J. Yang, and W. Shang. Studying Duplicate Logging Statements and Their Relationships With Code Clones. *IEEE Transactions on Software Engineering*, 48(7):2476–2494, July 2022. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2021.3060918. URL https://ieeexplore.ieee.org/document/9360483/.

X. Liu, T. Jia, Y. Li, H. Yu, Y. Yue, and C. Hou. Automatically Generating Descriptive Texts in Logging Statements: How Far Are We? In B. C. D. S. Oliveira, editor, *Programming Languages and Systems*, volume 12470, pages 251–269. Springer International Publishing, Cham, 2020. ISBN 978-3-030-64436-9 978-3-030-64437-6. doi: 10.1007/978-3-030-64437-6_13. URL https://link.springer.com/10.1007/978-3-030-64437-6_13. Series Title: Lecture Notes in Computer Science.

Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Which variables should i log? *IEEE Transactions on Software Engineering*, 47(9):2012–2031, 2021. doi: 10.1109/TSE.2019.2941943.

J. Long. Software reuse antipatterns. *SIGSOFT Softw. Eng. Notes*, 26(4):68–76, jul 2001. ISSN 0163-5948. doi: 10.1145/505482.505492. URL https://doi.org/10.1145/505482.505492.

J. Lu, D. S. Rosenblum, T. Bultan, V. Issarny, S. Dustdar, M.-A. Storey, and D. Zhang. Roundtable: The Future of Software Engineering for Internet Computing. *IEEE Software*, 32(1):91–97, Jan. 2015. ISSN 0740-7459. doi: 10.1109/MS.2015.15. URL http://ieeexplore.ieee.org/document/7030178/.

L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze. Troubleshooting Transiently-Recurring errors in production systems with Blame-Proportional logging. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 321–334, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/luo.

I. Macia Bertran, A. Garcia, and A. von Staa. An exploratory study of code smells in evolving aspect-oriented systems. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 203–214, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306058. doi: 10.1145/1960275.1960300. URL https://doi.org/10.1145/1960275.1960300.

M. Marron. Log++ logging for a cloud-native world. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 25–36, Boston MA USA, Oct. 2018. ACM. ISBN 978-1-4503-6030-2. doi: 10.1145/3276945.3276952. URL https://dl.acm.org/doi/10.1145/3276945.3276952.

N. Medvidovic, J. Garcia, D. Popescu, and G. Edwards. Identifying architectural bad smells. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 255–258, Los Alamitos, CA, USA, mar 2009. IEEE Computer Society. doi: 10.1109/CSMR.2009.59. URL `https://doi.ieeecomputersociety.org/10.1109/CSMR.2009.59`.

T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue. PADLA: A Dynamic Log Level Adapter Using Online Phase Detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 135–138, Montreal, QC, Canada, May 2019. IEEE. ISBN 978-1-72811-519-1. doi: 10.1109/ICPC.2019.00029. URL `https://ieeexplore.ieee.org/document/8813251/`.

H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 282–285, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312042. doi: 10.1145/2351676.2351724. URL `https://doi.org/10.1145/2351676.2351724`.

F. Palma and N. Mohay. A study on the taxonomy of service antipatterns. In *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, pages 5–8, 2015. doi: 10.1109/PPAP.2015.7076848.

K. Patel, J. Faccin, A. Hamou-Lhadj, and I. Nunes. The sense of logging in the Linux kernel. *Empirical Software Engineering*, 27(6):153, Nov. 2022. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-022-10136-3. URL `https://link.springer.com/10.1007/s10664-022-10136-3`.

G. Rong, S. Gu, H. Shen, H. Zhang, and H. Kuang. How Do Developers' Profiles and Experiences Influence their Logging Practices? An Empirical Study of Industrial Practitioners. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 855–867, Melbourne, Australia, May 2023. IEEE. ISBN 978-1-66545-701-9. doi: 10.1109/ICSE48619.2023.00080. URL `https://ieeexplore.ieee.org/document/10172853/`.

S. Saini, N. Sardana, and S. Lal. Logger4u: Predicting debugging statements in the source code. In *2016 Ninth International Conference on Contemporary Computing (IC3)*, pages 1–7, Noida, India, Aug. 2016. IEEE. ISBN 978-1-5090-3251-8. doi: 10.1109/IC3.2016.7880255. URL `http://ieeexplore.ieee.org/document/7880255/`.

J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. D. Nascimento, M. F. Freitas, and M. G. D. Mendonça. A systematic review on the code smell effect. *Journal of Systems and Software*, 144:450–477, Oct. 2018. ISSN 01641212. doi: 10.1016/j.jss.2018.07.035. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121218301444`.

T. Sharma and D. Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, Apr. 2018. ISSN 01641212. doi: 10.1016/j.jss.2017.12.034. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121217303114`.

T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901761. URL `https://doi.org/10.1145/2901739.2901761`.

V. S. Sharma and S. Anwer. Performance antipatterns: Detection and evaluation of their effects in the cloud. In *2014 IEEE International Conference on Services Computing*, pages 758–765, 2014. doi: 10.1109/SCC.2014.103.

B. Shen, T. Shan, and Y. Zhou. Improving logging to reduce permission Over-Granting mistakes. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 409–426, Anaheim, CA, Aug. 2023. USENIX Association. ISBN 978-1-939133-37-3. URL `https://www.usenix.org/conference/usenixsecurity23/presentation/shen-bingyu-logging`.

C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, 2000.

E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia. A systematic literature review on bad smells–5 w's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 2021. doi: 10.1109/TSE.2018.2880977.

D. Spencer and J. J. Garrett. *Card sorting: designing usable categories*. Rosenfeld Media, Brooklyn, N.Y, 2009. ISBN 978-1-933820-02-6. OCLC: ocn325125041.

K.-J. Stol, P. Ralph, and B. Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 120–131, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884833. URL `https://doi.org/10.1145/2884781.2884833`.

G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

D. Tovarnak, A. Vaekova, S. Novak, and T. Pitner. Structured and Interoperable Logging for the Cloud Computing Era: The Pitfalls and Benefits. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 91–98, Dresden, Germany, Dec. 2013. IEEE. ISBN 978-0-7695-5152-4. doi: 10.1109/UCC.2013.30. URL `http://ieeexplore.ieee.org/document/6809344/`.

A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

A. Vetro, L. Ardito, G. Procaccianti, M. Morisio, et al. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *Proceedings of ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, 2013.

W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, Big Sky Montana USA, Oct. 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629587. URL `https://dl.acm.org/doi/10.1145/1629575.1629587`.

N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik. 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)*, pages 61–70, Madrid, ES, May 2021. IEEE. ISBN 978-1-66543-869-8. doi: 10.1109/ICSE-SEIP52600.2021.00015. URL `https://ieeexplore.ieee.org/document/9402136/`.

K. Yao, G. B. De Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 127–138, Berlin Germany, Mar. 2018. ACM. ISBN 978-1-4503-5095-2. doi: 10.1145/3184407.3184416. URL `https://dl.acm.org/doi/10.1145/3184407.3184416`.

R. K. Yin. *Case study research : design and methods*. Applied social research methods series vol. 5 5 515259. Sage, Los Angeles, 5th ed. edition, 2014. ISBN 978-1-4522-4256-9.

D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, Hollywood, CA, Oct. 2012a. USENIX Association. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/yuan.

D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112, Zurich, June 2012b. IEEE. ISBN 978-1-4673-1066-6 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227202. URL http://ieeexplore.ieee.org/document/6227202/.

D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1), feb 2012c. ISSN 0734-2071. doi: 10.1145/2110356.2110360. URL https://doi-org.proxy.bnl.lu/10.1145/2110356.2110360.

Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering*, 24(6):3394–3434, Dec. 2019. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-019-09687-9. URL http://link.springer.com/10.1007/s10664-019-09687-9.

T. Zhang, H. Qiu, G. Castellano, M. Rifai, C. S. Chen, and F. Pianese. System log parsing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(8):8596–8614, 2023. doi: 10.1109/TKDE.2022.3222417.

J. Zhao, Y. Tang, S. Sunil, and W. Shang. Studying and complementing the use of identifiers in logs. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 97–107, 2023. doi: 10.1109/SANER56733.2023.00019.

X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. The Game of Twenty Questions: Do You Know Where to Log? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 125–131, Whistler BC Canada, May 2017a. ACM. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103001. URL https://dl.acm.org/doi/10.1145/3102980.3103001.

X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 565–581, Shanghai China, Oct. 2017b. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132778. URL https://dl.acm.org/doi/10.1145/3132747.3132778.

J. Zhaoxue, L. Tong, Z. Zhenguo, G. Jingguo, Y. Junling, and L. Liangxiong. A Survey On Log Research Of AIOps: Methods and Trends. *Mobile Networks and Applications*, 26(6):2353–2364, Dec. 2021. ISSN 1383-469X, 1572-8153. doi: 10.1007/s11036-021-01832-3. URL https://link.springer.com/10.1007/s11036-021-01832-3.

C. Zhi, S. Deng, J. Han, and J. Yin. Towards automatic detection and prioritization of pre-logging overhead: a case study of hadoop ecosystem. *Automated Software Engineering*, 29(1):11, May 2022. ISSN 0928-8910, 1573-7535. doi: 10.1007/s10515-021-00317-7. URL https://link.springer.com/10.1007/s10515-021-00317-7.

R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj. MobiLogLeak: A Preliminary Study on Data Leakage Caused by Poor Logging Practices. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 577–581, London, ON, Canada, Feb. 2020. IEEE. ISBN 978-1-72815-143-4. doi: 10.1109/SANER48275.2020.9054831. URL `https://ieeexplore.ieee.org/document/9054831/`.

J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to Log: Helping Developers Make Informed Logging Decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 415–425, Florence, Italy, May 2015. IEEE. ISBN 978-1-4799-1934-5. doi: 10.1109/ICSE.2015.60. URL `http://ieeexplore.ieee.org/document/7194593/`.

A Other logging issues

The survey identified eight logging related issues that did not fall under the three main categories. For completeness, these issues are presented in Table 8.

Table 8: Issues classified as "Something else".

Issue name	**Description**	**Implications**	**References**
Human or machine readable logs	Logs that are meant for humans can be difficult for machines to understand and vice versa. The target audience of the logs should be decided beforehand	The logs can become both machine- and human-unfriendly	Yuan et al. (2012a); Bogatinovski and Kao (2023); Jia et al. (2018)
Where to log	Determining where logging statements should be inserted in the source code	Can crash the program, create too much or too little logging, and hinder the comprehensibility of the source code	He et al. (2018); Li et al. (2020b); Yuan et al. (2012a); Zhu et al. (2015); Jia et al. (2018); Chen and Jiang (2022)
What to log	Determining what is relevant and should be logged in a logging statement	Can create too much or too little logging	Liu et al. (2021)
Cost of logging (money)	Logging costs money as the code and log files need to be developed, maintained, and stored	Insufficient logging takes resources from other aspects of a project	Li et al. (2021a); Lu et al. (2015); Fu et al. (2014); Zhaoxue et al. (2021)
Logging tools	Developers need tools to manage the large amount of logs.	The lack of tools makes developers to use more time to produce a high-quality log.	He et al. (2022); Yuan et al. (2012b); Fu et al. (2014); Lu et al. (2015)
Log file management	Issues related to managing the log files. For example, access control issues and managing the locations the log files are saved	Difficulty in finding files or a developer might not have access to the log files	He et al. (2022); Bijvank et al. (2013)
Security compliance	Ensuring the logging meets security measurements. This includes auditing, adding logging needed for potential forensics investigations, and ensuring the logs are not prone to vulnerabilities.	It takes a lot of resources from other tasks	Chen and Jiang (2022); Rong et al. (2023); Karumanchi and Squicciarini (2014)
Validating and verifying logs	Validating and verifying logs is difficult	The logs are not as reliable as they could be	Gu et al. (2023); Zhaoxue et al. (2021)

B Other logging tools

Our survey identified several logging related tools, which either did not fall under the category of detection (RQ2) or repair (RQ3) tool or did not focus on a log smell. Table 9 describes the excluded tools. The tools are divided based on their type. Tools marked with “LS” are logging systems, i.e., they propose a way of logging rather focusing on a single issue. “D” indicates a detection tool and “R” a repair tool.

Table 9: Tools presented in the papers which present a whole logging system (LS) or either detect (D) or repair (R) logging issues related to other aspects of logging.

Type	Tool name	Description	Source
LS	AUDIT	A blame-proportional logging system for troubleshooting transiently-recurrent problems in cloud-based production systems. The developer writes declarative triggers, specifying what to log and on what misbehavior, without specifying where to collect the logs	Luo et al. (2018)
LS	Cinque2009	A set of rules, to be followed at design time, specifically conceived to improve the quality of logged failure data and to ease the coalescence of redundant or equivalent data.	Cinque et al. (2009)
LS	Cinque2012	A rule-based approach to make logs effective for analyzing software failures. Leverages artifacts produced at system design time to create a set of rules to formalize the placement of logging code.	Cinque et al. (2013)
LS	JEL	Experimental logging mechanism. Produces monitoring data (free of natural language messages) that can be processed in an automated and interoperable way without extra effort from the developer.	Tovarnak et al. (2013)
LS	Log++	A logging system, that supports near zero-cost for disabled log statements, low cost lazy-copying for enabled log statements, selective persistence of logging output, unified control of logging output across different libraries, and DevOps integration for use with modern cloud-based deployments.	Marron (2018)
LS	Log2	A cost-aware "whether to log" logging mechanism.	Ding et al. (2015)
D	Fu2014	A classifier using type information and contextual information predicting whether to log for a code snippet (proof of concept tool)	Fu et al. (2014)
D	LACC	Predict location and severity level of logging statements at the method level	Gholamian and Ward (2020)
D	Li2020	A deep learning framework to automatically suggest logging locations at the block level.	Li et al. (2020b)
D	Log4Perf	An automated approach that provides suggestions of where to insert logging statement with the goal of monitoring web-based systems’ software performance.	Yao et al. (2018)
D	LogAdvisor	A logging suggestion tool which automatically learns the common logging practices on where to log from existing logging instances and further leverages them for actionable suggestions to developers.	Zhu et al. (2015)
D	Logger4u	A machine learning based framework for predicting logging of a new if-block.	Saini et al. (2016)
D	LogIm	An ensemble and threshold-based machine-learning model predicting need for logging in if- and catch-blocks.	Lal et al. (2016a)
D	LogOptPlus	A machine learning based tool to help optimizing the number of log statements in the source code for if-blocks and catch-blocks.	Lal et al. (2016c)
D	MobiLog-Leak	An approach that identifies log statements in deployed apps that leak sensitive data.	Zhou et al. (2020)
D	SECLOG	Tool using static analysis to automatically help developers find missing access-deny log locations and identify relevant information at the log location.	Shen et al. (2023)
D	TempoLo	Automatically detect the issues of temporal inconsistencies between logging and code.	Ding et al. (2023)
R	K9	A logging tool that automatically inserts logging code to trace inter-thread data dependencies.	Kubota et al. (2020)
R	Log20	A tool that determines a near optimal placement of log printing statements under the constraint of adding less than a specified amount of performance overhead.	Zhao et al. (2017b)
R	LogEnhancer	Tool for reducing the burden of failure diagnosis by enhancing the information that programmers should have captured when writing log messages.	Yuan et al. (2012c)
R	Zhao2017	A simple approach that automates INFO-logging stament placement	Zhao et al. (2017a)

Table 9 continued

Type	Tool name	Description	Source
R	Zhou2020	A log enhancement approach which automatically identifies logging points that reflect anomalous behavior during system fault creating machine-designed logs for machines to automatic fault diagnosis	Jia et al. (2018)