A Survey on Automated Log Analysis for Reliability Engineering

SHILIN HE, Microsoft Research
PINJIA HE, Department of Computer Science, ETH Zurich
ZHUANGBIN CHEN, TIANYI YANG, YUXIN SU, and MICHAEL R. LYU, Department of Computer Science and Engineering, The Chinese University of Hong Kong

Logs are semi-structured text generated by logging statements in software source code. In recent decades, software logs have become imperative in the reliability assurance mechanism of many software systems because they are often the only data available that record software runtime information. As modern software is evolving into a large scale, the volume of logs has increased rapidly. To enable effective and efficient usage of modern software logs in reliability engineering, a number of studies have been conducted on automated log analysis. This survey presents a detailed overview of automated log analysis research, including how to automate and assist the writing of logging statements, how to compress logs, how to parse logs into structured event templates, and how to employ logs to detect anomalies, predict failures, and facilitate diagnosis. Additionally, we survey work that releases open-source toolkits and datasets. Based on the discussion of the recent advances, we present several promising future directions toward real-world and next-generation automated log analysis.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software creation and management**.

Additional Key Words and Phrases: log, log analysis, logging, log compression, log parsing, log mining.

ACM Reference Format:
Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2021. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 1, 1 (June 2021), 37 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In the recent decades, modern software, such as search engines, instant messaging apps., and cloud systems, has been increasingly integrated into our daily lives and becomes indispensable. Most of these software systems are expected to be available on a 24×7 basis. Any non-trivial downtime can lead to significant revenue loss, especially for large-scale distributed systems [52, 53, 172]. For example, in 2017, a downtime in Amazon led to a loss of 150+ million US dollars [173]. Thus, the reliability of modern software is of paramount importance [130].

The work was supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002) and the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14210717). This work was mainly done when Shilin He was a Ph.D. student at the Chinese University of Hong Kong.
Authors' addresses: S. He, Microsoft Research, No. 5, Danling Street, Haidian District, Beijing, 100080, China; email: shilin.he@microsoft.com; P. He (corresponding author), Department of Computer Science, ETH Zurich, Universitätstrasse 6, 8092 Zürich, Switzerland; email: pinjia.he@inf.ethz.ch; Z. Chen, T. Yang, Y. Su, MR. Lyu, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong; email: {zbchen,lyu,yxsu,tyyang}@cse.cuhk.edu.hk.

0360-0300/2021/6-ART $15.00
https://doi.org/10.1145/1122445.1122456

```
1 public void setTemperature(Integer temperature) {
2   // ...
3   logger.debug("Temperature set to {}. Old temperature was {}.", t, oldT);
4   if (temperature.intValue() > 50) {
5     logger.info("Temperature has risen above 50 degrees.");
6   }
7 }
```

⇓

```
1 0 [setTemperature] DEBUG Wombat - Temperature set to 61. Old temperature was 42.
2 0 [setTemperature] INFO Wombat - Temperature has risen above 50 degrees.
```

Fig. 1. An example of logging statements by SLF4J and the generated logs.

Software logs have been widely employed in a variety of reliability assurance tasks, because they are often the only data available that record software runtime information. Additionally, logs also play an indispensable role in data-driven decision making in industry [149]. In general, logs are semi-structured text printed by logging statements (*e.g.*, `printf()`, `logger.info()`) in the source code. For example, in Fig. 1, the two log messages are printed by the two logging statements in the source code. The first few words (*e.g.*, "Wombat") of the log messages are decided by the corresponding logging framework (*e.g.*, SLF4J) and they are in structured form. On the contrary, the remaining words (*e.g.*, "50 degrees") are unstructured because they are written by developers to describe specific system runtime events.

A typical log analysis management framework is illustrated by the upper part of Fig. 2. Particularly, traditional logging practice (*e.g.*, which variables to print) mainly relies on developers' domain knowledge. During system runtime, software logs are collected and compressed as normal files using file compression toolkits [110] (*e.g.*, WinRAR). Additionally, developers leverage the collected logs in various reliability assurance tasks (*i.e.*, log mining), such as anomaly detection. Decades ago, these processes were based on specific rules specified by the developers. For example, to extract specific information related to a task (*e.g.*, thread ID), developers need to design regex (*i.e.*, regular expression) rules for automated log parsing [76]. The traditional anomaly detection process also relies on manually constructed rules [66]. These log analysis techniques were effective at the beginning because most of the widely-used software systems were small and simple.

However, as modern software has become much larger in scale and more complex in structure, traditional log analysis that is mainly based on ad-hoc domain knowledge or manually constructed and maintained rules becomes inefficient and ineffective [146]. This brings four major challenges to modern log analysis. (1) In many practices, while a number of senior developers in the same group share some best logging practices, most of the new developers or developers from different projects write logging statements based on domain knowledge and ad-hoc designs. As a result, the quality of the runtime logs varies to a large extent. (2) The volume of software logs has increased rapidly (*e.g.*, 50 GB/h [138]). Consequently, it is much more difficult to manually dig out the rules (*e.g.*, log event templates or anomalous patterns). (3) With the prevalence of Web service and source code sharing platforms (*e.g.*, Github), software could be written by hundreds of global developers. Developers who should maintain the rules often have no idea of the original logging purpose, which further increases the difficulty in manual maintenance of their rules. (4) Due to the wide adoption of the agile software development concept, a new software version often comes in a short-term manner. Thus, corresponding logging statements update frequently as well (*e.g.*, hundreds of new logging statements per month [178]). However, it is hard for developers to manually update the rules.

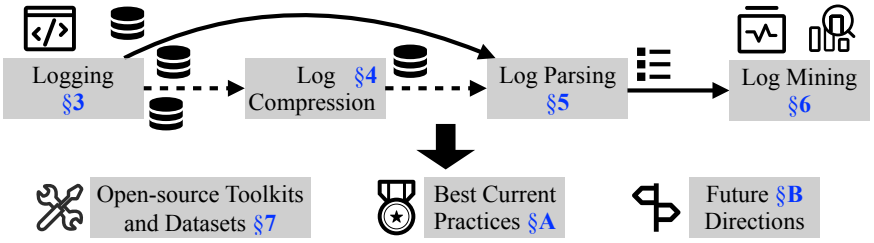

Fig. 2. An overall framework for automated log analysis.

To address these challenges, a great amount of work has been accomplished by both researchers and practitioners in recent decades. In particular, starting from 2003, a line of research efforts have been contributed to automated rule construction and critical information extraction from software logs, including the first pieces of work of log parsing [174], anomaly detection [174], and failure prediction [157]. In addition, in the same year, Hätönen *et al.* [70] proposed the first log specific compression technique. Later, many empirical studies were conducted as the first steps towards some difficult problems in automated log analysis, including the first study on failure diagnosis by Jiang *et al.* [83] in 2009, the first exploration of logging practice by Yuan *et al.* [189] in 2012, and the first industrial study on logging practice by Fu *et al.* [58] in 2014. Recently, machine learning and deep learning algorithms have been widely adopted by the state-of-the-art (SOTA) papers, such as the deep learning-based "what-to-log" approach [102] in logging practice and Deeplog [46] in anomaly detection. Besides machine learning, parallelization has been employed in various recent papers, such as Logzip [110] in log compression and POP [72] in log parsing.

These extensive studies on automated log analysis across multiple core directions have largely boosted the effectiveness and efficiency of systematic usage of software logs. However, the diversity and richness of both the research directions and recent papers could inevitably hinder the non-experts who intend to understand the SOTA and propose further improvements. To address this problem, this paper surveys 158 papers in the last 23 years across a variety of topics in log analysis. The papers under exploration are mainly from top venues in three related fields: software engineering (*e.g.*, ICSE), system (*e.g.*, SOSP), and networking (*e.g.*, NSDI). Thus, the readers can obtain a deep understanding of the advantages and limitations of the SOTA approaches, as well as taking a glance at the existing open-source toolkits and datasets. In addition, the insights and challenges summarized in the paper can help practitioners understand the potential usage of the automated log analysis techniques in practice and realize the gap between academy and industry in this field. We present crucial research efforts on automated log analysis from the following seven perspectives as illustrated in Fig. 2:

- Logging: Section 3 introduces approaches that automate or improve logging practices, including *where-to-log*, *what-to-log*, and *how-to-log*.
- Log compression: Section 4 presents approaches to compress software logs in runtime.
- Log parsing: Section 5 discusses how to automatically extract event templates and key parameters from software logs.
- Log mining: Section 6 introduces what automated log mining techniques can do to enhance system reliability. We focus on three main tasks: *anomaly detection*, *failure prediction*, and *failure diagnosis*.
- Open-source toolkits and datasets: Papers providing open-source toolkits and datasets, which facilitate log analysis research, are presented in Section 7.

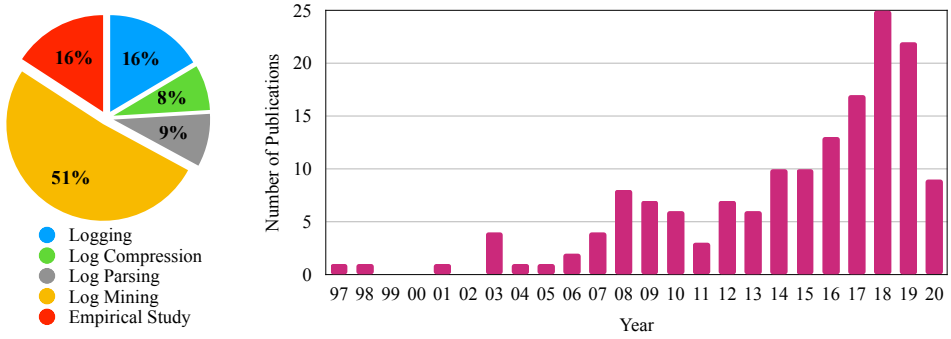

Fig. 3. Paper distribution on each research topic and the associated evolution trend from 1997 to 2020.

- Best current practices: Supplementary A presents some common log practices in industry, which might benefit the log analysis research and industrial deployment.
- Future directions: Supplementary B discusses open challenges and promising future directions that can push this field forward beyond current practices.

2 SURVEY METHODOLOGY

To systematically collect the publications for conducting this survey, we maintained a repository for the automated log analysis literature. We first searched relevant papers in online digital libraries and extended the repository by manually inspecting all references of these papers. The repository is now available online.[1]

To begin with, we searched several popular online digital libraries (*e.g.*, IEEE Xplore, ACM Digital Library, Springer Online, Elsevier Online, Wiley Online, and ScienceDirect) with the following keywords: "log", "logging", "log parsing", "log compression", "log + anomaly detection", "log + failure prediction", "log + failure diagnosis". The "log" term is broadly quoted in various domains, such as the query log in database, web search log in recommendation, and the logarithm function in mathematics. Hence, to precisely collect the set of papers of our interest, we mainly focused on regular papers published in top venues (*i.e.*, conferences and journals) of relevant domains, including ICSE, FSE, ASE, TSE, TOSEM, EMSE, SOSP, OSDI, ATC, NSDI, TDSC, DSN, ASPLOS, and TPDS. Then, we manually inspected each reference of these papers to collect additional publications that are related to the survey topics. In this paper, we focused on publications studying the reliability issues with software system logs, while publications of other topics are beyond the scope of this survey, such as logs for security and privacy [92], software change logs [23], kernel logs [148], and process mining logs [176].

In total, we collected 158 publications in automated log analysis area spanning from year 1997 to 2020. Fig. 3 (on the right part) shows the histogram of annual publications during the period. We can find that automated log analysis has been continuously and actively investigated in the past two decades. In particular, we can observe steady growth in the number of publications since 2006, indicating that the log analysis field has attracted an increasing amount of interest since then.

Furthermore, we classify these publications into five categories by research focus: logging, log compression, log parsing, log mining, and empirical study. The distribution is presented on the left part of Fig. 3. Note that some papers may address more than one research focuses, in which case we categorized the paper based on its major contribution. As expected, Fig. 3 shows that a large portion (around a half) of research efforts were devoted to log mining. The reasons are two-fold:

[1] https://github.com/logpai/awesome-log-analysis

First, log mining is a comprehensive task that consists of many sub-tasks (*e.g.*, anomaly detection, failure diagnosis). Each sub-task has a huge space for research exploration and thereby attracts many research studies on it; Second, research areas other than log mining are relatively mature. Besides, logging is also extensively studied in the past decades since it is more closely related to industrial practices. In addition, many log research tasks are empirical studies, and we discussed them in different sections respectively.

Furthermore, we were aware of several existing surveys on logs analysis. Oliner *et al.* [146] briefly reviewed sixteen log analysis papers and pointed out some challenges in 2012. Since then, the log analysis area has been actively studied, which impels a more comprehensive and systematic log survey paper. Other existing surveys mainly studied logs and their use in the security area. Khan *et al.* [89] focused on logs in the cloud computing environment and the logs were leveraged for identifying and preventing suspicious attacks. Zeng *et al.* [193] studied the logging mechanisms and security issues in three computer operating systems. A recent publication [94] reviewed clustering based approaches in cyber security applications. Different from these studies, our survey mainly targets the automated log analysis for tackling the general reliability issues.

3 LOGGING

Logging is the task of constructing logging statements with proper description and necessary program variables, and inserting the logging statements to the right positions in the source code. Logging has attracted attention from both academia [139, 149, 158, 180] and industry [9, 18, 58, 149, 189, 194] across a variety of application domains because logging is a fundamental step for all the subsequent log mining tasks.

3.1 Logging Mechanism and Libraries

3.1.1 Logging Mechanism. Logging mechanism is the set of logging statements and their activation code implemented by developers or a given software platform [149]. Fig. 1 shows two example logging statements and the collected logs during the execution of the program. The logging statement at line 3 is executed every time the `setTemperature` method is called, with no specific activation code. The logging statement at line 5 is controlled by the activation code `if (temperature.intValue() > 50)` at line 4. According to the data collected by [149], the most widely-adopted coding pattern that is used to activate the logging statements is `if (condition) then log error()`.

3.1.2 Logging Libraries. To improve flexibility, industrial developers often utilizes logging libraries [20, 149], which are software components that facilitate logging and provide advanced features (*e.g.*, thread-safety, log archive configuration, and API separation). Toward this end, a lot of open-source logging libraries have been developed (*e.g.*, Log4j [116], SLF4J [166], AspectJ [5], spdlog [167]).

3.2 Challenges for Logging

Logging in a software system is usually decided by an empirical process in the development phase [149]. In general, logging practice, *i.e.*, how developers conduct the task of logging, is scarcely documented or regulated by strict standard, such as the logging mechanism and APIs [9]. Thus, logging relies heavily on human expertise [58, 71, 146, 149, 160]. In the following, we summarize three main challenges for automated logging, which also align with the taxonomy mentioned by Chen and Jiang [17]: *where-to-log*, *what-to-log*, and *how-to-log*. Under this categorization, every problem exhibits aspects that represent the primary concerns of the actual practice of logging. Accordingly, we summarize three major aspects, *i.e.*, *diagnosability*, *maintenance*, and *performance*, as presented in Table 1.

3.2.1 where-to-log. Where-to-log is about determining the appropriate location of logging statements. Although logging statements provide rich information and can be inserted almost everywhere, excessive logging results in performance degradation [17] and incurs additional maintenance overhead. In addition, it is challenging to diagnose problems by analyzing a large volume of logs as most logs are unrelated to the problematic scenarios [85]. On the other hand, insufficient logging will also impede the logs' diagnosability. For example, an incomplete sequence of logs may hinder the reproduction of precise execution paths [199]. Therefore, developers need to be circumspect in their choices of where-to-log.

3.2.2 what-to-log. What-to-log is about providing sufficient and concise information within the three major components of a logging statement, *i.e.*, verbosity level, static text, and dynamic content. Mis-configured verbosity level has similar consequences with inappropriate logging points. As developers typically filter logs according to the verbosity levels, under-valued verbosity levels may result in missing or ignored log messages while over-valued verbosity levels lead to overwhelming log messages [85]. When composing a snippet of logging code, the static text should be concise and the dynamic content should be coherent and up-to-date. Poorly written static text and inconsistent dynamic content could affect the subsequent diagnosis and maintenance activities [86, 111, 136, 196].

3.2.3 how-to-log. How-to-log is the "design pattern" and maintenance of logging statements systematically. Most software testing techniques focus on verifying the quality of feature code, but a few papers [17, 18, 68, 162, 189] pay attention to the quality and anti-patterns in the logging code. Most industrial and open source systems choose to scatter logging statements across the entire code base, intermixing with feature code [17], which also hardens the maintenance of logging code.

3.3 Logging Approaches

Numerous solutions have been proposed to address the challenges mentioned in Section 3.2. Table 1 summarizes existing studies along with corresponding problems and aspects. Each row represents a challenge. The approaches fall into three categories: *static code analysis*, *machine learning*, and *empirical study*. A bunch of early work utilized static code analysis to analyze logging in a program without executing the source code. Machine learning-based approaches focus on learning from data. By concentrating on the inherent statistical properties of existing logging statements, learning-based approaches automatically give suggestions on improving the logging statements. In the remainder of this section, we discuss solutions by their aspects.

3.3.1 Diagnosability. Logs are valuable for investigating and diagnosing failures. However, a logging statement is only as helpful as the information it provides. The research on this aspect aims at (1) understanding the helpfulness of logs for failure diagnosis and (2) making logs informative for diagnosis.

(1) Understanding the helpfulness of logs for failure diagnosis. This is of great importance because logs are widely adopted for failure diagnosis. According to a survey [58] involving 54 experienced developers in Microsoft, almost all the participants agreed that "logging statements are important in system development and maintenance" and "logs are a primary source for problem diagnosis". Fu *et al.* [58] also studied the types of logging statements in industrial software systems by source code analysis, and summarized five types of logging snippets, *i.e.*, assertion-check logging, return-value-check logging, exception logging, logic-branch logging, and observing-point logging. Besides, Fu *et al.* [58] further demonstrated the potential feasibility of predicting where to log. Shang *et al.* [163] conducted the first empirical study to provide a taxonomy for user inquiries of logs. They identified five types of information that were often sought from log lines by practitioners, *i.e.*, meaning, cause, context, impact, and solution. Shang *et al.* [163] were also the first to associate the development

Table 1. Summary of logging approaches.

Problems	Aspects	Objectives
where-to-log	Diagnosability	Suggest appropriate placement of logging statements into source code [31, 32, 102, 183, 188, 198, 199, 203] ; Study logging practices in industry [58, 97].
	Performance	Minimize or reduce performance overhead [44, 188].
what-to-log	Diagnosability	Enhance existing logging code to aid debugging [190]; Suggest proper variables and text description in log [71, 98, 111].
	Maintenance	Determine whether a logging statement is likely to change in the future [87]; Characterize and detect duplicate logging code [103].
	Performance	Study the performance overhead and energy impact of logging in mobile app [26, 194]; Automatically change the log level of a system in case of anomaly [141].
how-to-log	Diagnosability	Characterize the anti-patterns in the logging code [163]; Optimize the implementation of logging mechanism to facilitate failure diagnosis [129].
	Maintenance	Characterize and detect the anti-patterns in the logging code [17, 18, 68, 99, 101, 189]; Characterize and prioritize the maintenance of logging statements [86]; Study the relationship between logging characteristics and the code quality [19, 162]; Propose new abstraction or programming paradigm of logging [90, 115].
	Performance	Optimize the compilation and execution of logging code [182].

knowledge at present in various development repositories (*e.g.*, code commits and issues reports) with the log lines and to assist practitioners in resolving real-life log inquiries. In addition, Li *et al.* [97] highlighted the feasibility of guiding developers' logging practice with topic models by investigating six open source systems.

(2) Making logs informative for failure diagnosis. As printed logs are often the only run-time information source for debugging and analysis, the quality of log data is critically important. *LogEnhancer* [190] made the first attempt to systematically and automatically augment existing logging statements in order to reduce the number of possible code paths and execution states for developers to pinpoint the root cause of a failure. Zhao *et al.* [198, 199] followed the idea of *LogEnhancer* and proposed an algorithm capable of completely disambiguating the call path of HDFS requests. Yuan *et al.* [188] found that the majority of unreported failures were manifested via a generic set of error patterns (*e.g.*, system call return errors) and proposed the tool *Errlog* to proactively add pattern-specific logging statements by static code analysis.

As modern software becomes more complex, where-to-log has become an important but difficult decision, largely limited to the developer's domain knowledge. Around 60% of failures due to software faults do not leave any trace in logs, and 70% of the logging pattern aims to detect errors via a checking code placed at the end of a block of instructions [31, 32]. Cinque *et al.* [32] concluded that the traditional logging mechanism has limited capacity due to the lack of a systematic error

model. They further formalized the placement of the logging instruction and proposed to use system design artifacts to manually define *rule-based logging* which utilizes error models about what cause errors to fail. Zhu *et al.* [203] made an important first step towards the goal of "learning to log". They proposed a logging recommendation tool, *LogAdvisor*, that learns the common logging rules on where-to-log from existing code via training a classifier and further leverages it for informative and viable recommendations to developers. Yao *et al.* [183] leveraged a statistical performance model to suggest the need for updating logging locations for performance monitoring. Li *et al.* [102] proposed a deep learning framework to suggest where-to-log at the block level. Li *et al.* also concluded that there might be similar rules regarding the implementation of logging mechanism across different systems and development teams, which agreed with the industrial survey by Pecchia *et al.* [149].

The lack of strict logging guidance and domain-specific knowledge makes it difficult for developers to decide what-to-log. To address this need, Li *et al.* [98] employed ordinal regression model to suggest proper verbosity level based in software metrics. He *et al.* [71] conducted the first empirical study on the usage of natural language in logging statements. They showed the global (*i.e.*, in a project) and local (*i.e.*, in a file) repeatability of text descriptions. Furthermore, they demonstrated the potential of automated description text generation for logging statements. Liu *et al.* [111] proposed a deep learning-based approach to recommend variables in logging statements by learning embeddings of program tokens. In order to troubleshoot transiently-recurring problems in cloud-based production systems, Luo *et al.* [129] put forward a new logging mechanism that assigns a blame rank to methods based on their likelihood of being relevant to the root cause of the problem. With the blame rank, logs generated by a method over a period of time are proportional to how often it is blamed for various misbehavior, thus facilitating diagnosis.

3.3.2 Maintenance. The maintenance of logging code has also attracted researchers' interest. The research on this aspect aims at (1) characterizing the maintenance and detecting anti-patterns of logging statements and (2) proposing new abstractions of logging.

(1) Characterizing the maintenance and detecting anti-patterns of logging statements. Anti-patterns in logging statements are bad coding patterns that undermine the quality and effectiveness of logging statements and increases the maintenance effort of projects. Many papers [18, 19, 68, 162] performed empirical studies to reveal the link between logs and defects. These papers observed a positive correlation between logging characteristics and post-release defects. Therefore, practitioners should allocate more effort to source code files with more logging statements.

Yuan *et al.* [189] made the first attempt to conduct a quantitative characteristic study of how developers log within four pieces of large open-source software. They described common anti-patterns and provided insights into where developers spend most of their efforts in modifying the log messages and how to improve logging practice. They further implemented a prototype checker to verify the feasibility of detecting unknown problematic statements using historical commit data. Chen and Jiang [17] and Hassani *et al.* [68] both studied the problem of how-to-log by characterizing and detecting the anti-patterns in the logging code. The analysis [17] of well-maintained open-source systems revealed six anti-patterns that are endorsed by developers. Chen and Jiang [17] then encoded these anti-patterns into a static code analysis tool to automatically detect anti-patterns in the source code. Li *et al.* [103] developed an automated static analysis tool, *DLFinder*, to detect duplicate logging statements that have the same static text description.

Just like feature code, logging code updates with time [87]. Moreover, logging statements are often changed without consideration for other stakeholders, resulting in sudden failures of log analysis tools and increased maintenance costs for such tools. Pecchia *et al.* [149] reviewed the industrial practice in the reengineering of logging code. Kabinna *et al.* [86] empirically studied the migration of logging libraries and the main reasons for the migration. Li *et al.* [99] derived

and used a set of measures to predict whether a code commit requires log changes. Kabinna *et al.* [87] later examined the important metrics for determining the stability of logging statements and further leveraged learning-based models (random forest classifier and Cox proportional hazards) to determine whether a logging statement is likely to remain unchanged in the future. Their findings were helpful to build robust log analysis tools by ensuring that these tools relied on logs generated by more stable logging statements. Li *et al.* [101] designed a tool to learn log revision rules from logging context and modifications and recommend candidate log revisions.

(2) Proposing new abstractions of logging. Maintaining logging code along with feature code has proven to be error-prone [18, 68, 162]. Hence, additional logging approaches [90, 115] have been proposed to resolve this issue. Kiczales *et al.* [90] proposed a new programming paradigm that improves the modularity of the logging code. To tackle the ordering problem of logs in distributed systems, Lockerman *et al.* [115] introduced *FuzzyLog* that featured strong consistency, durability, and failure atomicity.

3.3.3 Performance. The intermixing nature of the logging code and feature code usually incurs performance overhead, storage cost, and development and maintenance efforts [26, 44, 141, 190, 194]. Tools like *LogEnhancer* [190], *Errlog* [188], *Log2* [44], and *INFO-logging* [198] all took performance into consideration while dealing with diagnosability and maintenance issues. Mizouchi *et al.* [141] proposed a dynamical adjusting verbosity level to record irregular events while reducing performance overhead. Yang *et al.* [182] proposed *NanoLog*, a nanosecond scale logging system that achieved relatively low latency and high throughput by moving the workload of logging from the runtime hot path to the post-compilation and execution phases of the application. Chowdhury *et al.* [26] were the first to explore the energy impact of logging in mobile apps. Zeng *et al.* [194] conducted a case study that characterized the performance impact of logging on Android apps.

4 LOG COMPRESSION

After collecting logs by executing logging statements during runtime, logs are stored for failure diagnosis or sensitive operations auditing. Then, how to store logs efficiently becomes a challenging problem. Since large-scale software systems run on a 24×7 basis, the generated log size has been huge (*e.g.*, 50 gigabytes per hour [138]). Besides, many logs require the long-term storage for identifying duplicate problems and mining failure patterns from historical logs [2, 43, 191]. Auditing logs that record sensitive user operations are often stored for two years and even more to track system misuse in future. Archiving logs in such a huge volume for a long period brings inconceivably heavy burden to storage space, electrical power, and network bandwidth for transmission. To tackle these problems, a line of research has been focusing on log compression, which aims to remove the redundancy in a large log file and reduce its storage consumption.

4.1 Challenges for Log Compression

Practically, log compression can be achieved by various approaches. The most straightforward way is to reduce the amount of logging statements in the source code or to set a less verbose log level (e.g., change from "INFO" to "ERROR") [98] in runtime. Although effective, this method leads to information loss in logs, which impedes the log-based troubleshooting. Besides, it is a common practice in existing logging libraries (*e.g.*, log4j) to compress log files by some general file compressors, such as gzip. However, the method is not tailored for the semi-structured log format, making it very ineffective in log compression. The structured fields (*i.e.*, constant parts) are often repetitive across many raw logs, which consume a lot of storage space and should be handled specifically. To address these issues, a tailored log compression algorithm is highly in demand.

Table 2. Summary of log compression approaches.

Methods		GC	Target Data	Scalability	Heuristics
Bucket	Balakrishnan *et al.* [7]	Yes	Blue Gene/L logs	High	Yes
	Skibinski *et al.* [165]	Yes	All logs	Low	No
	Hassan *et al.* [67]	No	Telecom logs	High	Yes
	Christensen *et al.* [27]	Yes	All logs	High	Yes
	Feng *et al.* [55]	Yes	All logs	Middle	No
Dictionary	Deorowicz *et al.* [42]	No	Apache web log	Middle	Yes
	Lin *et al.* [106]	No	Structured Logs	High	Yes
	Mell *et al.* [134]	Yes	Structured Logs	High	No
	Racz *et al.* [153]	Yes	Web log	High	Yes
Statistics	Hatonen *et al.* [70]	No	Structured Logs	High	Yes
	Meinig *et al.* [133]	No	All logs	High	Yes
	Liu *et al.* [110]	Yes	Structured Logs	Low	No
Industry	Splunk [168]	Yes	All logs	Low	No
	ElasticSearch [49]	Yes	All logs	Low	No

4.2 Characteristics for Log Compression

An ideal log compressor should achieve a high compression ratio while imposing short compression and decompression time. The compression ratio denotes the ratio between the file sizes before and after the compression. A higher compression ratio indicates that less storage space is consumed and the compression algorithm is more effective. Additionally, log compression algorithm takes a certain amount of time to compress and decompress the file respectively, which is supposed to be as short as possible. In this paper, we mainly explored 12 existing log compressing approaches and several empirical studies. As shown in Table 2, to clearly demonstrate the advantages and disadvantages of these approaches, we list several characteristics that concern the log compressors based on the following descriptions in the surveyed papers. *(1) Use of general compressors (GC).* Some log compressors utilize the general compressor as the backend technique after reformatting the logs. Specifically, these reformattings can transform raw logs to better formats that achieve a higher compression ratio. *(2) General applicability* denotes whether the log compressor can be generally applicable to various log data formats. A general log compressor shows better utility in different scenarios. For example, as shown in the "target data" of Table 2, the log compressor proposed in [42] was specialized for Apache web log while other log compressors [27] can be applied to different log data types. *(3) Scalability.* Since logs are often of a great volume in practice, the efficiency of log compression is crucial to the practical employment. A highly scalable compressor could save both the compression and decompression time, which thereby saves the log query time and supports real-time analysis. *(4) Heuristics.* The deployment would be more efficient and smoother if the log compressor requires little prior knowledge, *e.g.*, heuristic rules on log formats and preprocessing procedures. The less heuristics a log compressor needs, the more generally applicable it would be.

4.3 Log Compression Approaches

According to the techniques employed above, there are three categories of log compression methods: (1) *bucket-based compression* that divides the log data into different blocks and compresses each block in parallel; (2) *dictionary-based compression* that builds a dictionary for fields in the log

and replaces strings by referring to the dictionary; and (3) *statistics-based compressors* that apply complex statistical models to find correlations between logs before compression. Note that some log compression methods may involve two or more techniques, in which case we took the major compression technique as their category.

4.3.1 Bucket-based Compression. Bucket-based compression methods break the log data into different blocks by some log-specific characteristics and then compresses these blocks in parallel. For instance, many logs show the strong temporal locality or high similarity in certain fields, therefore, these logs can be gathered and then compressed. To tackle the problem that log data were often heterogeneous with varying patterns over time, Christensen *et al.* [27] proposed to partition the log data into different buckets by leveraging the temporal locality and then compress these buckets in parallel. Hassan *et al.* [67] adopted a similar idea for telecom logs; however the objective was log event sequence instead of log messages. The log data is firstly broken into equal sized periods and compressed separately. Different from other studies, they use the compressed log event sequences to identify noteworthy usage scenarios for operational profile customization.

Besides, the bucket partition often serves as an initial step towards achieving a high compression ratio. For example, LogPack [165] was a multi-tiered log compression method, in which each tier addressed one notion of redundancy. The first tier handled the local redundancy between neighboring lines. The second tier handled the global repetitiveness of tokens while the third tier handled all the remaining redundancy by employing a general compressor. Similarly, Balakrishnan *et al.* [7] first observed a number of trends in the Blue Gene/L system logs and proposed several solution to compress the logs accordingly. For example, one trend was that most columns in adjacent logs tended to be the same. At last, the generic compression utilities were applied to further compress the log file. Hence, they utilized a variant of delta encoding method which compared the log with preceding logs and encoded the differences only. In addition, Multi-level Log Compression (MLC) [55] divided logs with redundancy into different buckets by calculating the Jaccard similarity among logs. Then, the logs were condensed by a variant of delta encoding, followed by a general compressor to further improve the compression ratio.

4.3.2 Dictionary-based Compression. Dictionary-based compression removes the redundancy in log files by replacing repetitive strings with references to a dictionary. For example, some frequent strings (*e.g.*, IP addresses) can be mapped to a condensed string in the dictionary. Lin *et al.* [106] presented a column-wise independent compression for structured logs, which was also very similar to the work presented in [42, 153] though their focuses were web logs. The general idea of these methods is as follows: At first, it splits a log entry into several columns by its fields. After observing common properties in columns (*e.g.*, common prefixes and suffixes), the method builds a dictionary-based model with Huffman code to compress each column separately. Many general compression techniques can then be applied to achieve a higher compression ratio, including move-to-front coding, phrase sequence substitution, etc. Similarly, Mell *et al.* [134] proposed a multi-step method for log compression. It first separated and sorted logs according to particular properties (*e.g.*, value of the first column), and then it created a dictionary where logs are stored by hashing the field name, followed by serializing the dictionary and compressing logs with a general compressor.

4.3.3 Statistics-based Compression. Statistics-based methods realize the log compression by building a statistical model to identify possible redundancy in the log data. Unlike the above two categories of methods which are supported by manually defined compression logic, the statistics-based compression automatically mines the compression rules. Hatonen *et al.* [70] demonstrated a Comprehensive Log Compression (CLC) method to dynamically characterize and combine log data entries. Particularly, the method first identified frequently occurring patterns from dense log

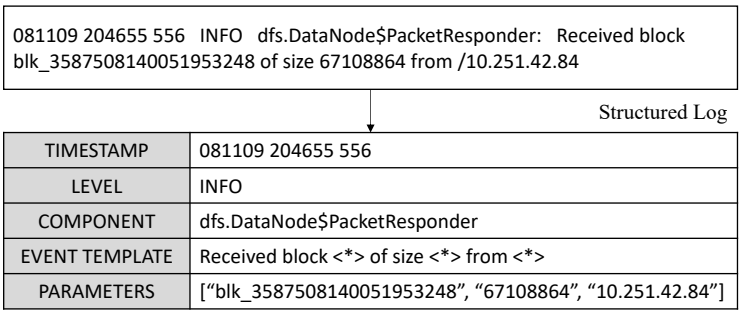

Fig. 4. A log parsing example.

data by frequent pattern mining, and then linked patterns to the data as a data directory. Meinig *et al.* [133] proposed an approach adopted from the rough set in the uncertainty theory. It treated the log file as a decision table with removable attributes identified by a one-time analysis of log data. Specifically, attributes that could roughly express each other are then collapsed to one representative attribute. The proposed method belongs to the set of lossy compression approaches which allow the information loss after compression. Lossy compression usually sacrifices the information completeness in raw logs for the high efficiency of log compression. Liu *et al.* [110] proposed the Logzip method based on log parsing introduced in Section 5. Logzip first automatically extracted log templates from raw log messages by a statistical log parsing model. Then Logzip structuralized log templates and other information into intermediate representations, which were finally fed into a general compressor.

4.3.4 Others. Recently, Yao *et al.* [184] provided an empirical study on comparing the performance of general compressors on compressing log data against natural language data. Although the study was targeted on general compressors instead of specialized log compressors, their findings could potentially guide the design of tailored log compressors. Besides, Otten *et al.* [147] presented a systematic review on general compression techniques, based on which they investigated the use of compression for log data reduction and the use of semantic knowledge to improve data compression. Different from the classic log compression, Hamou-Lhadj et al. [65] proposed to summarize the content of logs for the purpose of understanding software system behaviors. In detail, their approach removes implementation details such as utility information from the execution traces by ranking system components under a well-defined metric.

5 LOG PARSING

After log collection, log messages will be input into different downstream log mining tasks (*e.g.*, anomaly detection) for further analysis. However, most of the existing log mining tools [75, 180] require structured input data (*e.g.*, a list of structured log events or a matrix). Thus, a crucial step of automated log analysis is to parse the semi-structured log messages into structured log events.

Fig. 4 presents a log parsing example, where the input is a log message collected from Hadoop Distributed File System (HDFS) [180]. A log message is composed of message header and message content. The message header is determined by the logging framework and thus it is relatively easy to extract, such as verbosity levels (*e.g.*, "INFO"). In contrast, it is difficult to extract key information from the message content because it is mainly written by developers in free-form natural language. Typically, the message content contains *constants* and *variables*. Constants are the fixed text written by the developers (*e.g.*, "Received") and describe a system event, while variables are the values of the program variables which carry dynamic runtime information. The goal of log parsing is to

distinguish between *constants* and *variables*. All the constants form the *event template*. The output of a log parser is a structured log message, containing an event template and the key parameters.

5.1 Challenges for Log Parsing

As a key component, log parsing has become an appealing selling-point [118, 123, 154] of many industrial log management solutions [204] (*e.g.*, Splunk [168]). However, these industrial solutions only support common log types such as Apache logs [154]. When parsing general log messages, they rely on regexs and ad-hoc scripts provided by developers. These scripts separate log messages into different groups, where log messages in the same group have the same event template. In modern software engineering, even with these existing industrial solutions, log parsing is still a challenging task due to three main reasons: (1) the large volume of logs and thus the great effort on manual regex construction; (2) the complexity of software and thus the diversity of event templates; and (3) the frequency of software updates and thus the frequent update of logging statements.

5.2 Log Parser Characteristics

We explore 15 automated log parsing approaches. To provide a clear glimpse of these parsers, we focus on three main characteristics: mode, coverage, and preprocessing. We follow the definition introduced by Zhu *et al.* [204]. Note that different from their work, which was an evaluation study providing benchmarks and open-source implementation of existing log parsers, this survey focuses on the methodological comparison of these parsers. In addition, while parsing rules can be constructed by analyzing source code [180], we focus on parsers that only utilize logs as input.

Mode. Mode is the most important log parsing characteristic to consider. Depending on the usage scenarios of parsing, log parsers can be categorized into two modes: offline and online. Offline log parsers require all the log messages beforehand and parse log messages in a batch manner. To cope with the frequent software update, developers need to periodically re-run the offline parser to obtain the newest event templates. In contrast, online parsers parse the log messages in a streaming manner, which work seamlessly with the downstream log mining tasks.

Coverage. We denote coverage as the capability of a log parser to match all input log messages with event templates. Note that this is orthogonal to whether the matched event templates are correct or not. In Table 3, “Partial" indicates the parser can only parse part of the log messages (*i.e.*, some logs will be matched with no event templates). For example, SLCT [174] matches a log message with an event template only if the log message contains a specific “frequent pattern" and thus leaving the remaining log messages unparsed. Therefore, SLCT achieves “Partial” coverage. “Partial" coverage might lead to the neglection of crucial system anomalies.

Preprocessing. Preprocessing removes some *variables* or replaces them by *constants* based on domain knowledge. For example, IP addresses (*e.g.*, 10.251.42.84) are typical variables in cloud systems’ log messages. This step requires some manual efforts (*i.e.*, constructing regexs for these variables).

5.3 Offline Log Parsing Approaches

Frequent Pattern Mining. SLCT (Simple Logfile Clustering Tool) [174] is the first research paper on automated log parsing. SLCT conducted two passes in total to obtain associated words. In the first pass, SLCT counted the occurrence of all the tokens and marked down the frequent words. These frequent words were utilized in the second pass to find out associated frequent words. Finally, for a log message, if it contained a pattern of associated frequent words, these words would be regarded as constants and employed to generate event templates. Otherwise, the log message would

Table 3. Summary of log parsing approaches.

	Methods	Coverage	Preprocessing	Technique
Offline	SLCT [174]	Partial	No	Frequent pattern mining
	AEL [84]	All	Yes	Heuristics
	LKE [57]	All	Yes	Clustering
	LFA [143]	All	No	Frequent pattern mining
	LogSig [171]	All	No	Clustering
	IPLoM [131, 132]	All	No	Iterative partitioning
	LogCluster [175]	Partial	No	Frequent pattern mining
	LogMine [64]	All	Yes	Clustering
	POP [72]	All	Yes	Iterative partitioning
	MoLFI [137]	All	Yes	Evolutionary algorithms
Online	SHISO [142]	All	No	Clustering
	LenMa [164]	All	No	Clustering
	Spell [45]	All	No	Longest common subsequence
	Drain [73, 74]	All	Yes	Heuristics
	Logram [35]	All	Yes	Frequent pattern mining

be placed into an outlier cluster without matched event templates. *LFA* [143] adopted a similar strategy as SLCT. Differently, LFA could cover all the log messages.

Clustering. LogCluster [175] is similar to SLCT [174]. Differently, LogCluster allowed variable length of parameters in between via a clustering algorithm. Thus, compared with SLCT, LogCluster is better at handling log messages of which the parameter length is flexible. For example, “Download Facebook and install" and "Download Whats App and install" have the same event template “Download <*> and install" while the length of the parameter (*i.e.*, an app name) is flexible. *LKE* (Log Key Extraction) [57] was developed by Microsoft. LKE adopted a hierarchical clustering algorithms with a customized weighted edit distance metric. Additionally, the clusters were further partitioned by heuristic rules. *LogSig* [171] was a more recent clustering-based parser than LKE. Instead of directly clustering log messages, LogSig transformed each log message into a set of word pairs and clustered logs based on the corresponding pairs. *LogMine* [64] adopted an agglomerative clustering algorithm. It was implemented in map-reduce framework for better efficiency.

Heuristics. AEL [84] employed a list of specialized heuristic rules. For example, for all the pairs like “word=value," AEL regarded the “value" as a variable and replaced it with a “$v" symbol.

Evolutionary Algorithms. MoLFI [137] formulate log parsing as a multi-objective optimization problem and propose an evolutionary algorithm-based approach. Specifically, MoLFI employs the Non-dominated Sorting Genetic Algorithm II [41] to search for a Pareto optimal set of event templates. Compared with other log parsers, the strength of MoLFI is that it requires little parameter tuning effort because the four parameters required by MoLFI has effective default values. However, because of the adoption of evolutionary algorithm, the MoLFI is slower than most of the parsers.

Iterative Partitioning. IPLoM [131, 132] contained three steps and partitioned log messages into groups in a hierarchical manner. (1) Partition by log message length. (2) Partition by *token position*. The position containing the least number of unique words is “token position". Partitioning was

conducted according to the words in the token position. (3) Partition by mapping. Mapping relationships were searched between the set of unique tokens in two token positions, which were selected using a heuristic criterion. *POP* [72] is a parallel log parser that utilizes distributed computing to accelerate the parsing of large-scale software logs. POP can parse 200 million HDFS log messages in 7 mins, while most of the parsers (*e.g.*, LogSig) failed to terminate in reasonable time.

5.4 Online Log Parsing Approaches

Clustering. SHISO [142] is the first online log parsing approach. SHISO used a tree-form structure to guide the parsing process, where each node was correlated with a log group and an event template. The numbers of children nodes in all the layers were the same and were manually configured beforehand. During the parsing process, SHISO traversed the tree to find the most suitable log group by comparing the log message and the event templates in the corresponding log groups. SHISO is sensitive to path explosion and thus its efficiency is often unsatisfactory. *LenMa* [164] is similar to SHISO. LenMa encodes each log message into a *length vector*, where each dimension records the number of characters of a token. For example, “Receive a file." would be vectorized as [7, 1, 5]. During parsing, LenMa would compare the length vectors of the log messages.

Longest Common Subsequence. Similar to SHISO and LenMa, *Spell* [45] maintained a list of log groups. To accelerate the parsing process, Spell utilized specialized data structures: prefix tree and inverted index. In addition, Spell provided a parallel implementation.

Heuristics. Drain [74] maintained log groups via the leaf nodes in the tree. The internal nodes of the tree embedded different heuristic rules. The extended version of Drain [73] was based on a directed acyclic graph that allowed log group online merging. In addition, it provided the first automated parameter tuning mechanism for log parsing.

Frequent Pattern Mining. Logram [35] is the current state-of-the-art parser. Different from the existing approaches that count frequent tokens, Logram considered frequent n-gram. The core insight of Logram is: frequent n-gram are more likely to be constants. Note that Logram assumed developers had some log messages on hand to construct the dictionary.

6 LOG MINING

Log mining employs statistics, data mining, and machine learning techniques for automatically exploring and analyzing large volume of log data to glean meaningful patterns and informative trends. The extracted patterns and knowledge could guide and facilitate monitoring, administering, and troubleshooting of software systems. In this section, we first elaborate on challenges encountered in log mining (Section 6.1). Then, we describe the general workflow of log mining (Section 6.2). Finally, we introduce three major log mining tasks for reliability engineering, including *anomaly detection* (Section 6.3), *failure prediction* (Section 6.4), and *failure diagnosis* (Section 6.5). Other relevant studies with relatively lesser popularity are laid out in Section 6.6.

6.1 Challenges of Log Mining

Traditionally, engineers perform simple keyword search (such as "error", "exception", and "failed") to mine suspicious logs that might be associated with software problems, *e.g.*, component failures. Some rule-based tools [66, 151, 155] have been developed to detect software problems by comparing logs against a set of manually defined rules which describe normal software behaviors. However, due to the ever-increasing volume, variety, and velocity of logs produced by modern software, such approaches fall short for being labor-intensive and error-prone. Moreover, suspicious logs are often

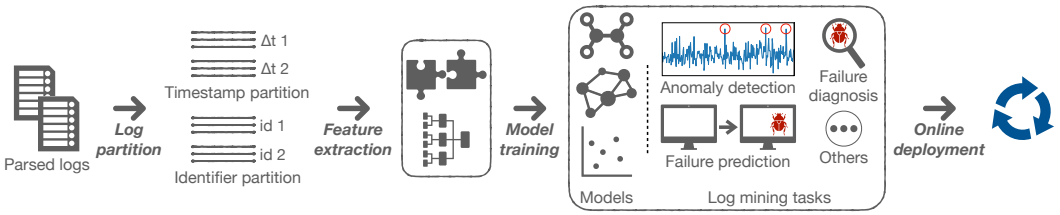

Fig. 5. A general workflow of log mining.

overwhelmed by logs generated during software normal executions. Manually sifting through a massive amount of logs to identify failure-relevant ones is like finding a needle in a haystack.

Additionally, inspecting logs for software troubleshooting often requires engineers to possess descent knowledge about the software. However, modern software systems usually consist of many components developed by different engineers, leading to the generation of heterogeneous logs, which makes troubleshooting beyond the ability of a single engineer. Moreover, due to the high complexity of modern software systems, failures could stem from various sources of software and hardware issues. Examples include software bugs, hardware damage, OS crash, service exception, etc. In addition, promptly pinpointing to the root cause by inspecting logs highly relies on engineers' expertise and experience. However, such knowledge is often not well accumulated, organized, and documented. Therefore, sophisticated ways to conduct automatic log mining are in high demand.

6.2 A General Workflow of Log Mining

The general workflow of log mining is illustrated in Fig. 5, which mainly consists of four steps, *i.e.*, log partition, feature extraction, model training, and online deployment.

6.2.1 Log Partition. Modern software systems often adopt a microservice architecture and comprise a large number of modules operating in a multi-threaded environment. Different microservices or modules often aggregate their execution logs into a single log file, which hinders automated log mining. To tackle the problem, interleaved logs should be partitioned into different groups, each of which represents the execution of individual system tasks. The partitioned log group is the basic unit in extracting features before constructing log mining models. Moreover, studies [76] show that the way to partition logs impacts the log mining performance. In the literature, we observe that two elements often serve as the log partitioner, namely, *timestamp* and *log identifier*.

Timestamp. It records the occurrence time of each log, which is a fundamental feature supported by many logging libraries, such as Log4j. Through regexs, the timestamp with different formats can be easily extracted from raw logs in log parsing phase (Section 5). In general, two strategies are often adopted to conduct timestamp-based log partition, *i.e.*, fixed window and sliding window. Fixed window has a predefined window size, which means the time span or time interval (*e.g.*, 30 minutes) used to split chronologically sorted logs. Extended from the fixed window, sliding window allows the overlapping between two consecutive fixed windows. The sliding window has two attributes, *i.e.*, window size and step size. The step size indicates the forwarding distance of the window alone the time axis to generate log partitions, which is often smaller than the window size.

Log Identifier. It is a token identifying a series of related operations or message exchanges of the system. For instance, HDFS logs adopt *block_id* to record the operations (*e.g.*, allocation, replication, and deletion) on a specific block. Common log identifiers include *user ID*, *task/session/job ID*, *variable/component name*, etc., which can be extracted by log parsing. Compared to timestamp, log identifier is a clearer and more definite signal for partitioning logs. Therefore, many research studies [57] employed the log identifier for its ability in distinguishing logs of different task

executions. However, the log identifiers might be non-unique in representing distinct system entities in different logs, such as virtual machines, physical machines, and networks [186]. This is because, for example, identifiers may not be propagated and synchronized across different services, one thread or process may serve more than one request through multiplexing, etc. Therefore, dedicated algorithms have been developed to address this problem and we elaborate on them later.

6.2.2 Feature Extraction. To analyze logs automatically, textual logs in a log partition should be transformed into appropriate formats that could fit machine learning algorithms. Through reviewing the literature, we identified two categories of log-based features, namely, numerical feature and graphical feature. Particularly, numerical feature is the mainstreaming feature which is widely used in the log analysis community.

Numerical Feature. It represents log's statistical properties including numerical and categorical fields that can be directly extracted from logs. It conveys the information of a log partition into a numerical vector representation. Particularly, numerical features employed by most of the existing work are similar and share some typical forms. In the following, we compactly summarize four types of such features.

- *Log event sequence*: A sequence of log events recording system's activities. Particularly, each element can simply be the log event ID or log embedding vector, *e.g.*, learned by word2vec algorithms [140].
- *Log event count vector*: A feature vector recording the log events occurrence in a log partition, where each feature denotes a log event type and the value counts the number of occurrence.
- *Parameter value vector*: A vector recording the value of parameters (*i.e.*, variables extracted by log parsing) that appear in logs.
- *Ad-hoc features*: A set of relevant and representative features extracted from logs, which are defined using domain knowledge on the object software system and problem context. For example, Zhou *et al.* [202] manually identified various types of features from system trace logs, including configuration (*e.g.*, memory limit, CPU limit), resource (*e.g.*, memory consumption, CPU consumption), etc. These features profile the typical health states of a system.

Graphical Feature. To discover the hierarchical and sequential relations (*e.g.*, dependency and co-occurrence) between system components and events with logs, the graphical feature usually produces a directed graph model characterizing system behaviors, *e.g.*, the execution path of a process. The graphical features serve as the foundation for a variety of downstream log mining tasks (such as monitoring [186, 200] and diagnosis [3, 8, 145]). For example, log-based behavioral differencing [62] can identify system executions that are derived from system normal behaviors, which has various applications in system evolution, testing, and security [8]. In this section, we briefly introduce the algorithms used by existing work for graphical feature extraction, which can be roughly categorized into two lines of work.

The first line of work leverages the objects identified in logs, *e.g.*, process ID and system component name, to construct graphs for system state monitoring. As objects often demonstrate complicated hierarchical relations, it usually needs a more sophisticated algorithm for log partition. For instance, to represent execution structure and object hierarchy in logs, Zhao *et al.* [200] constructed a stack structure graph by considering the 1:1, 1:n, and n:m mappings among different objects. To tackle the challenge that unique identifiers are often unavailable, Yu *et al.* [186] proposed to group interleaved logs based on a common set of identifiers. Each log contains an identifier set that acts as a state node of the graph, and transitions are added by examining the subset and superset relations of different identifier sets.

The other line of work makes use of the underlying statistical distribution of log events, *e.g.*, the order of log events, the temporal and spatial locality of dependent log events, to build various graphs for system behavior analysis. Particularly, some work aims at recovering the exact behavioral model from log traces. Log partition therefore requires that identifiers can tie together the set of events associated with a program execution. For example, Amar *et al.* [3] presented two variants of k-Tails [13], namely 2KDiff and nKDiff, to build Finite State Machine (FSM) for log differencing. 2KDiff computes and highlights the differences between two log files containing a set of partitioned logs; while nKDiff is able to conduct the comparison for multiple log files at once. Later Bao *et al.* [8] extended their work by proposing s2KDiff and snKDiff, which take into consideration the frequencies of different behaviors. Busany *et al.* [14] studied the scalability problem of existing behavioral log analysis algorithms by extracting Finite State Automaton (FSA) models or temporal properties from logs. We refer readers to this paper for more related studies.

However, as previously discussed, a unique identifier for each execution trace is not always available, especially in distributed applications and systems. Therefore, some work attempts to mine behavioral models using interleaved logs. For example, Lou *et al.* [124] employed statistical inference to learn the temporal dependencies among log events, which is also studied in similar work including [11, 12]. Nandi *et al.* [145] computed the nearest neighbor groups to capture the temporal co-occurrence of log events more accurately. The resulted model is a program Control Flow Graph (CFG) spanning distributed components. Particularly, the correlation between two components is calculated using either the Jaccard similarity or the Bayesian conditional probability approach. Du *et al.* [46] described two methods to capture service executions by FSA models. The first method leverages the trained log anomaly detection model, whose predictions encode the underlying path of task execution. The second method builds a matrix where each entry represents the co-occurrence probability of two log keys appearing together within a predefined distance.

6.2.3 Model Training. In this process, appropriate algorithms are selected based on the problem at hand and selected models are trained based on the extracted features. A variety of machine learning algorithms have been proposed, and we leave more details in the following sections. This step is often conducted in an offline manner. Moreover, a fundamental assumption of various log mining tasks is that the majority of logs should exhibit patterns that conform to system's normal behaviors. For example, in anomaly detection, different models are trained to capture various patterns from different perspectives and used to detect anomalies that lack the desireable properties.

6.2.4 Online Deployment. Once the model is trained offline, it could be deployed to real-world software systems for various log mining tasks. For example, an anomaly detection model can be integrated into software products to detect malicious system behaviors and raise alarms in real time. To tackle the challenge of pattern change caused by system upgrades, several studies [46, 108] supported online update of a previously trained model to adapt to unprecedented log patterns.

6.3 Anomaly Detection

6.3.1 Problem Formulation. Anomaly detection is the task of identifying system anomalous patterns that do not conform to expected behaviors on log data. Typical anomalies often indicate possible error, fault or failure in software systems. In this section, we elaborate on existing research work in this field based on approaches adopted therein. As shown in Fig. 6, we categorize the approaches into two broad classes: traditional machine learning algorithms and deep learning models. The surveyed approaches are listed in Table 4 together with several interesting properties. Specifically, we summarize the algorithm/model and feature used by an approach and whether or not each approach is unsupervised and online. Particularly, unsupervised approaches do not require labels for model training; and an online approach is one that can process its input piece-by-piece in a

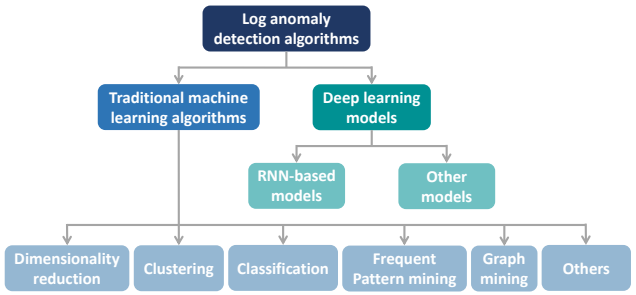

Fig. 6. A taxonomy of log anomaly detection algorithms.

Table 4. Summary of log anomaly detection approaches.

	Methods	Algorithm/Model	Feature	Unsupervised	Online
Traditional machine learning	Xu *et al.* [180]	PCA	★ †	Yes	No
	Lin *et al.* [108]	Clustering	∗	Yes	No
	He *et al.* [75]	Clustering	∗ ★	Yes	No
	Liang *et al.* [104]	SVM	‡	No	No
	Kimura *et al.* [91]	SVM	‡	No	No
	Xu *et al.* [179]	Frequent pattern mining	∗ ★	Yes	Yes
	Shang *et al.* [161]	Frequent pattern mining	∗	Yes	No
	Lou *et al.* [125]	Frequent pattern mining	★	Yes	No
	Farshchi *et al.* [54]	Frequent pattern mining	★	Yes	No
	Nandi *et al.* [145]	Graph mining	¶	Yes	No
	Lou *et al.* [124]	Graph mining	¶	Yes	No
	Yamanishi *et al.* [181]	Statistical model	∗	Yes	No
	He *et al.* [76]	Logistic regression	★	No	No
Deep learning	Du *et al.* [46]	LSTM model	∗ †	Yes	Yes
	Zhang *et al.* [196]	LSTM classification model	∗	No	No
	Meng *et al.* [136]	LSTM model	∗ ★	Yes	Yes
	Xia *et al.* [177]	LSTM-based GAN model	∗	Yes	Yes
	Lu *et al.* [128]	CNN model	∗	No	No
	Liu *et al.* [109]	Graph embedding model	¶	Yes	No

∗ Log event sequence, ★ Log event count vector, † Parameter value vector
‡ Ad hoc features, ¶ Graphical feature

streaming fashion. In this paper, we mainly focus on general types of anomalies that are natural imperfections of software systems and will directly impact system's reliability. External attacks and intrusions (such as cyber attacks and malicious activities) that are more relevant to system's security go out of the scope for this paper.

6.3.2 Traditional Machine Learning Algorithms. Traditional machine learning algorithms usually perform on top of features explicitly provided by practitioners, *e.g.*, log event count vector. Particularly, the anomaly detection task can be formalized into different types and solved with different algorithms such as clustering, classification, regression, etc.

Dimensionality Reduction. It transforms data of high dimension into a low-dimension representation such that some meaningful properties of the original data can be retained in the low-dimension space. Principal Components Analysis (PCA) is one of the most popular algorithms of this kind. By projecting data points to the first k principal components, anomalies can be identified if the projected distance is larger than a threshold. PCA was first applied by Xu *et al.* [179, 180] to mine system problems from console logs. Particularly, both log event count vector and parameter value vector are constructed and fed into a PCA model for anomaly detection.

Clustering. As an unsupervised method, clustering-based anomaly detection groups log-based feature vectors into different clusters such that vectors in the same cluster are more similar to each other (and as dissimilar as possible to vectors from other clusters). Clusters that contain very few data instances tend to be anomalous. For example, Lin *et al.* [108] proposed LogCluster, which clusters log sequences and recommends a representative sequence to assist developers quickly identify potential problems. The representative sequence is selected by calculating the cluster centroid. Furthermore, He *et al.* [75] proposed the Log3C framework to incorporate system KPIs into the identification of impactful problems in service systems. In particular, they proposed a cascading clustering algorithm to group numerous log sequences promptly. They finally use a multivariate linear regression model to identify impactful problems that lead to KPI degradation.

Classification. Anomaly detection with classification categorizes the log partition into either normal or anomalous type, where anomalous examples are derived from normal ones in terms of some statistical properties. Support Vector Machine (SVM) is a supervised classification method that is commonly used for log anomaly detection. In [104], Liang et al. vectorized log partitions by identifying six types of features, including events number in a time window, accumulated events number, etc. Based on these features, they applied four classification models for anomaly detection, *i.e.* including SVM and nearest neighbor predictor. In addition, Kimura et al. [91] proposed a log analysis method for proactive failure detection based on logs' characteristics, including frequency, periodicity, burstiness, and correlation with maintenance and failures. A SVM model with the Gaussian kernel is employed for detecting failures.

Frequent Pattern Mining. It aims to discover the most frequent item sets and sub-sequences in a log dataset that characterize system's normal behaviors. Data instances that do not conform to the frequent patterns will be reported as anomalies. The presence of specific log events and the order of log events can both constitute patterns. For example, Xu *et al.* [179] mined sets of log messages that often co-occur to detect abnormal execution traces in an online setting. Compared to offline approaches, online pattern matching can quickly identify benign system executions, thus striking a balance between accuracy and efficiency. Similarly, Lim *et al.* [105] searched for item sets that were commonly associated with different failure types. Such item sets can assist developers in predicting and characterizing failures. Other approaches mine the sequential patterns by log events to discover anomalies or bugs [126, 161].

Additionally, Lou et al. [125] were the first to consider mining invariants among log messages for system anomaly detection. Two kinds of invariants that model the relations between the number of different log messages are derived: (1) invariants in textual logs that describe the equivalence relation; and (2) invariants as a linear equation, which is the linear independence relation. Farshchi *et al.* [54] proposed a similar approach which mines the correlation and causation relations between log events and cloud system metric changes. Specifically, they employed a regression-based approach to learn a set of assertions modeling the linear relations. Anomaly detection is then conducted by monitoring log event streams and checking the compliance of metrics against the assertions.

Graph Mining. The set of approaches mainly employs graphical features, *i.e.*, various graph models (Section 6.2), to identify behavioral changes of complex systems, which can be used for early detection of anomalies and allowing proactive actions for correction. For example, Nandi *et*

al. [145] introduced a CFG mining approach to detect anomalous runtime behaviors of distributed applications from execution logs, including both sequence anomaly and distribution anomaly. A sequence anomaly is raised when an expected child is missing for a parent node within the given time interval; a distribution anomaly is raised when an edge probability is violated. In [57], Fu et al. modeled the execution behaviors of each system module as a state transition graph by learning an FSA from log sequences. Each transition in the learned FSAs corresponds to a log key. For each state transition, two types of information (*i.e.*, the time consumed and circulation number) are recorded and applied to detect two performance issues: low transition time and low transition loop. By employing the Gaussian distribution to model the state transition in a distributed system, low-performance transitions can be automatically identified by setting a proper threshold.

Other Statistical Models. There are some algorithms that do not belong to the above categories. For example, Yamanishi *et al.* [181] employed a mixture of Hidden Markov Models to monitor syslog behaviors. Particularly, the model is learned with an online discounting learning algorithm by dynamically selecting an optimal number of mixture components. Anomaly scores are assigned using universal test statistics whose threshold can be dynamically optimized. He *et al.* [76] employed a logistic regression model to detect anomalies. They adopted event count vectors as the feature and trained the model with a set of labeled data. A testing instance is declared as anomalous when the probability estimated by the logistic function is greater than 0.5. Nagaraj *et al.* [144] diagnosed performance issues for large-scale distributed systems by finding a set of inter-related log event occurrences and variable values that exhibit the largest divergence across logs sets. Specifically, they first separated logs into two sets according to some performance metrics (*e.g.*, runtime). Then, they recommended log events or state variables that contribute the most to the performance difference via t-tests.

6.3.3 Deep learning models. Deep learning uses a multiple-layer architecture (*i.e.*, neural networks) to progressively extract features from inputs with different layers addressing different levels of feature abstraction. Due to the exceptional ability in modeling complex relationships, neural networks are widely applied in log-based anomaly detection. In the literature, we observed a significant portion of papers adopting the Recurrent Neural Network (RNN) model and its variants. Hence, we classify the deep learning models into RNN-based models and other models.

RNN-based Models. Models belonging to the RNN family (such as LSTM model and GRU model) are commonly used to automatically learn the sequential patterns in log data. Anomalies are raised when log patterns deviate from the model's predictions. For example, Du *et al.* [46] proposed DeepLog, which utilizes an LSTM model to learn system's normal execution patterns by predicting the next log event given a sequence of preceding log events. However, some anomalies may manifest themselves as an irregular parameter value instead of a deviation from a normal execution path. Hence, DeepLog also applies an LSTM model for checking the validity of parameter value vectors. Many existing studies assume that the log data are stable over time and the set of distinct log events is fixed and known. However, Zhang *et al.* [196] found that log data often contain previously unseen log events or log sequences, demonstrating log instability. To tackle this problem, they proposed LogRobust to extract the semantic information of log events by leveraging off-the-shelf word vectors and then applied a bidirectional LSTM model to detect anomalies.

In terms of capturing log's semantics, Meng *et al.* [136] found existing word2vec models did not distinguish well between synonyms and antonyms. Therefore, they trained a word embedding model to explicitly consider the information of synonyms and antonyms. Meng *et al.* [135] further extended their work by proposing a semantic-aware representation framework for online log analysis. The issues of log-specific word embedding and out-of-vocabulary (OOV) are both addressed. Recently, Zuo *et al.* [205] combined the transaction-level topic modeling for learning the embedding of

logs, where a transaction is a group of logs sequentially occurring in a time window. To address the problem of insufficient labels, Chen *et al.* [24] applied transfer learning to share anomalous knowledge between two software systems. Specifically, they first trained an LSTM model on the data with sufficient anomaly labels to extract sequential log features, which were then fed into fully connected layers for anomaly classification. Next, the LSTM model was fine-tuned with logs from another system with limited labels, while the fully connected layers were fixed.

Other Deep Learning Models. Besides the RNN-based models, other model architectures also play a role in detecting anomalies with logs. For example, Xia *et al.* [177] proposed LogGAN, an LSTM-based Generative Adversarial Network (GAN). Like all GAN-style models, LogGAN comprises a generator and a discriminator. The generator attempts to capture the distribution of real training data and synthesizes plausible examples; while the discriminator tries to distinguish fake instances from real and synthetic data. Effort is also devoted to explore the feasibility of Convolutional Neural Networks (CNN) for anomaly detection. Specifically, Lu *et al.* [128] first utilized a word embedding technique to encode logs into two-dimension feature matrices, upon which CNN models with different filters were then applied for anomaly detection. Liu *et al.* [109] proposed log2vec, a graph embedding based method for cyber threat detection. Specifically, they first converted logs into a heterogeneous graph using heuristic rules and then learned the embedding of each log entry by a graph representation learning approach. Based on the embedding vectors, logs were grouped into clusters, whose size smaller than a threshold would be reported as malicious.

6.4 Failure Prediction

6.4.1 Problem Formulation. Anomaly detection aims to detect anomalous status or unexpected behavior patterns which may or may not cause failures. Differently, failure prediction attempts to proactively generate early warnings to prevent server failures, which often lead to unrecoverable states. The objective and role of anomaly detection and failure prediction are different yet both are important. Therefore, the exploration of failure prediction techniques is significant to reliability engineering. Generally, when software systems deviate from fulfilling a required system function [95], a failure occurs and it often shows human-perceivable symptoms. The failure could lead to unintended results and user dissatisfaction, especially for large-scale software systems. Traditional ways of failure management (such as anomaly detection) are mostly passive, which deal with failures after they have happened. In contrast, failure prediction aims to proactively predict the failure before it happens. A common practice is to leverage the valuable logs to proactively predict failures. For example, in [195], switch failures in data center networks are predicted from current status and curated historical hardware failure cases. Usually, input data of the predictive model are system logs, which record the system status, changes in configuration, and operational maintenance, etc.

According to the source of failures, failure prediction could mainly be categorized into two scenarios: a) Prediction of independent failures in homogeneous systems, *e.g.*, high-performance computing (HPC) systems. Most existing approaches focus on how to leverage the sequential information to predict the failure of each single component, *i.e.*, sequence-based methods. b) Prediction of outages caused by a collection of heterogeneous devices or components, which is widely observed in large-scale cloud systems. The mainstreaming approach tends to explore useful hints in the relationship between multiple heterogeneous components and makes prediction from relation-mining algorithms.

6.4.2 Homogeneous Systems. In homogeneous systems, the mainstreaming approaches are based on modeling the sequential information that represents the system status. A typical homogeneous system is large-scale supercomputers which may encounter faults (e.g., component failures) every

day. Exascale systems are expected to combat with higher fault rates due to enormous number of system components and higher workload [39]. However, triggering resilience-mitigating mechanisms is difficult since there are no obvious, well-defined failure indicators, which rely on a deeper understanding of the faults caused by hardware and software components. Therefore, logs become the most reliable information source to monitor the system health. Furthermore, to catch early warning signs of these failures, there are additional applications that scan through the monitoring data from the system logs and proactively generate alerts. Some of these alerts are generated from the manually defined thresholds on the collected system performance data (*e.g.*, CPU utilization rate) or error logs (*e.g.*, certain types of errors appearing too frequently within some time range). The objective of such alerts is to provide early warnings so that protective actions (*e.g.*, virtual machine migration or software rejuvenation) can be performed to mitigate or minimize the impact of such failures. In this context, efficient failure prediction via system log mining can enable proactive recovery mechanisms to increase reliability.

In homogeneous systems, component failures or node soft lock-ups typically lead to crashes of user jobs scheduled on the affected nodes, and they may cause undesired downtime. One approach to mitigate such problems is to predict node failures with a sufficient lead time in order to take proactive measures. Sahoo *et al.* [157] first addressed this problem by collecting system logs representing the components' health status. Then, several time series models are employed to predict the system health of each node through indication metrics, such as the percentage of system utilization, usage of network IO, and system idle time. Russo *et al.* [156] considered the log sequence as multi-dimensional vector and employed three kinds of support vector machines to predict defective log sequences. Das *et al.* [39] proposed Desh to identify failure indicators in each node with enhanced training and classification for failure event log chain. Desh is a deep learning-based method that contains three phases: (1) It first recognizes chains of log events leading to a failure. (2) Then it re-trains chain recognition of events augmented with expected lead times to failure; (3) Finally Desh predicts lead times during testing/inference deployment to predict which specific node fails in how many minutes. To speed up the recognition of the failure chain from log events, Das *et al.* [38] proposed a new node failure predicting method called Aarohi to extend their previous work to online learning setting. Aarohi first trains an offline deep learning model with log parsing, then utilizes grammar-based rules to provide online testing. Another approach from feature engineering tries to extract specific patterns about a certain time frame before a critical event. This anomaly pattern can serve as a proactive alerts. Klinkenberg *et al.* [93] adopted descriptive statistics and supervised machine learning techniques to create such proactive alert from the system logs. They trained a binary classification model to detect the potential node failure based on a given time sequence of monitoring data collected from each node. Berrocal *et al.* [10] used environmental logs to extract numerical indicators and conducted a Void Search (VS) algorithm on these numerical values for the failure prediction task.

6.4.3 Heterogeneous Systems. In heterogeneous systems, modeling the relationship among multiple components is the key, which is also the core part for the dominating relation mining algorithms. Different from homogeneous systems which focus on predicting failures based on raw signals sealed in the event logs or trace logs from single component, the prediction of critical failures (also called outage) in heterogeneous systems relies on the failure signals from log information collected from diversified system components. In current real-time heterogeneous systems, such as the cloud system, outage prediction is an important and challenging task to perform. On one hand, outages are critical system failures that could lead to severe consequences. On the other hand, outages occur without a significant alerting signals pattern, which makes it hard to predict. Moreover, the scope of impacting signals is a complex process to define. Therefore, compared to homogeneous

system settings, it is more valuable to detect node outages and distinguish regular internal failures from those caused by external factors, such as maintenance and human errors.

From the perspective of methodology, the advancement of failure prediction in heterogeneous systems relies on effective detection of the relationship between early alerting signals from system logs and node/component outage. Chen *et al.* [25] proposed AirAlert framework based on Bayesian network to find the conditional dependence between the alerting signal extracted from system logs and the outage. Then, given several alert signals, AirAlert employed gradient boosting tree method to conduct outage prediction. Lin *et al.* [107] designed MING framework to find the relationship between complex failure-indicating alerts and the outage from temporal and spatial features. They carefully designed a ranking model that combines the intermediate results from LSTM model capturing temporal signals and a Random Forest model incorporating spatial data. Zhou *et al.* [202] proposed MEPFL to narrow down the scope of failure prediction into microservice level. Based on a set of manual selected features defined on the system trace logs, MEPFL trains prediction models at both the trace level and the microservice level to predict three common types of microservice application faults: multi-instance faults, configuration faults, and asynchronous interaction faults.

6.5 Failure Diagnosis

6.5.1 Problem Formulation. Unlike anomaly detection and failure prediction that are usually formalized as a classification task, failure diagnosis targets to identify the underlying causes leading to a failure that has affected end users. It is often closely related to the root cause analysis. Specifically, although anomaly detection and failure prediction can pinpoint whether a problem occurs or will occur, there is a huge gap between the detection and removal of a problem or a failure. To completely resolve the problems, failure diagnosis is a crucial step, but the diagnosis process is notoriously expensive and inefficient. It is reported that failure diagnosis takes over 100 billion dollars, and developers spend more than half of their time on debugging [197]. Following the concept of error, fault, and failure, as defined in [95], failure diagnosis aims to identify the fault that has led to the user-perceived impairment in a software system. In the broad domain of failure diagnosis, log-based failure diagnosis is now a standard practice for software developers. However, failure diagnosis is very challenging. The complexity of modern software systems grows rapidly, where different services, software, and hardware are tightly coupled. It is too complex to correctly and efficiently disentangle the relations among the fault, failure, and human-observed symptoms. Moreover, as software systems become more mature, failures are becoming more and more hard to detect and diagnose, from perceptible software functionality issues to imperceptible problems [88], *e.g.*, performance issues [1, 138, 185].

To tackle these challenges, in recent decades, techniques to automate the diagnosis process have been widely developed, for example, using the informative logs [29, 82, 202]. Jiang *et al.* [83] provided one of the first characteristic studies of log-based problem troubleshooting on real-world cases. They concluded that problem troubleshooting is time-consuming and challenging, which can be significantly facilitated by logs. In addition, they appealed to engineers to employ automated methods to speedup the problem resolution time, which is also the focus of this survey. Likewise, Zhou *et al.* [201] empirically investigated the faults and debugging practices in microservice systems. The results show that proper tracing and visualization techniques can improve the diagnosis, which also suggest the strong needs for more intelligent log analysis. Other empirical studies include understanding failures with logs in high-performance computing (HPC) systems [48], cloud management systems [34], web servers [78] and industrial air traffic control system [30].

In this section, we review recent work on automated log-based failure diagnosis. Particularly, these research studies mainly focus on the large-scale software systems where failure diagnosis is burdensome, such as general distributed systems, storage systems, big data systems, microservice

systems. As the characteristic and functionality may vary, these systems demonstrate different failure behaviors. However, the overall methodology for failure diagnosis still shares similar techniques, which can be categorized into four types, *i.e.*, execution replay, model-based, statistics-based, and retrieval-based methods. There are also studies on diagnosing the hardware fault by traces (*e.g.*, [100]), which are beyond the scope of this survey.

6.5.2 Execution Replay Methods. Traditional rule-based methods heavily rely on a set of predefined rules (*e.g.*, in the format of "if-then") from the expert knowledge to diagnose failures. However, the methods cannot be well generalized to unseen failures that are not included in the rules. On the contrary, execution replay methods aim to automatically infer the execution flow from logs and trace back to the software system failure, as previously introduced in Section 6.2. The set of methods is human-interpretable and automated. Yuan *et al.* [187] proposed the SherLog, which analyzes source code by run-time logs to represent the detailed execution process of a failure. Specifically, SherLog infers both control and data value information of a failed execution to help developers understand the failure. Similarly, LogMap [16] first retrieves log messages from bug reports and then applies static analysis technique to identify corresponding logging lines in source code. At last, it traverses through logging lines to derive the potential code paths, which help reconstruct the execution path and assist the debugging process.

6.5.3 Model-based Methods. Model-based methods utilize logs to build the reference model (*e.g.*, execution path) for a software system and then check which log events violate the reference model. In short, it sets up standards for normal software system executions and diagnoses failures by detecting the possible inconsistency. The majority of existing model-based studies leverage the log information to reconstruct the system execution flow as a graph representation [6, 80, 81]. To achieve so, Jia et al. proposed to mine a time-weighted control flow graphs (TCFG) [81] and the service topology [80] from interleaved logs during the offline phase. In the online phase, a failure can be easily diagnosed by observing the deviation between execution log sequences and the mined graph models. After observing that log sequences generated by a normal execution are consistent across multiple runs, Tak *et al.* [170] proposed the LOGAN to capture normal log patterns by log grouping, log parsing, and log alignment. When a failure occurs, LOGAN highlights the divergence of current log sequence from a reference model and suggests the possible root cause. Different from these research tasks, to detect concurrency bugs in distributed systems, Lu *et al.* [126] proposed to mine logs from historical executions to uncover feasible but undertested log message orders. The log message orders represent possible execution flow that are likely to expose errors.

6.5.4 Statistics-based Methods. Since software systems generate logs to record normal and abnormal executions, it is intuitive to employ some statistical techniques (*e.g.*, statistical distribution, correlation analysis) to capture the relationships between logs and the consequent failures.

Chuah *et al.* [29] developed a diagnostics tool, FDiag, to parse the log messages and employs statistical correlation analysis to attribute the observed failure to a possible root cause. However, the diagnostic capability of the proposed method is limited to known failures. To overcome the difficulty, Chuah *et al.* [28] further extended their approach to an advanced FDiagV3 method, in which a PCA and ICA-based correlation approach is additionally employed. Differently, FDiagV3 can identify those unknown failures by automatically extracting the outlier issues. In [59], a three-step approach is designed to establish the causal dependency graph through log events, which helps identify the process that a failure occurs. The general idea resembles execution replay methods, but the causal dependency graph is statistically mined by grouping similar log events and utilizing the temporal order information. Similarly, Yu *et al.* [185] proposed to comprehend performance problem of device drivers in Windows. The method narrows down the diagnosis scope and pattern

mining by measuring the impact of suspicious components on performance and representing the behavior pattern with the signature set tuple. Then, thresholds are set to identify highly suspicious and high-impact components that are likely to cause performance problems.

Lu *et al.* [127] proposed to detect and diagnose anomalies in the Spark system. In complementary to previous studies, the proposed method builds statistical models from the data distribution of several task-related features. It then diagnoses the anomaly by setting threshold and analyzes the root causes with weighted factors. Similar to Xu et al. [180], SCMiner [192] was proposed to utilize a PCA method to detect abnormal system call sequences. Then, it maps the abnormal sequence to application functions by frequent pattern mining on system call traces. At last, SCMiner identifies and ranks buggy functions by matching with the function call traces. Likewise, CloudDiag [138] employs a statistical technique (*i.e.*, based on the data distribution) to identify the category of a fine-grained performance problem. Then, a fast matrix recovery algorithm, RPCA, is adopted to identify the root cause (*i.e.*, method invocations) for the performance failure.

6.5.5 Retrieval-based Methods. In practice, failures that previously occurred are valuable since they can aid developers in better diagnosing newly-occurred failures. Retrieval-based methods, as indicated by the name, retrieve similar failures in a knowledge base composed of failures in history or populated by injected faults in the test environment. Shang *et al.* [161] focused on diagnosing big data analytics applications in Hadoop system by injecting failures manually and analyzing the logs. Particularly, they proposed to uncover differences between pseudo and cloud deployments by log parsing, execution sequence recovery, and sequence comparison. Pham *et al.* [150] adopted a similar idea but they targeted on the general distributed system. Under the assumption that similar faults generate similar failures, we can locate the root cause of a reported failure by inspecting matched failures in the knowledge base. The method first reconstructs execution flows between system components, computes the similarity of the reconstructed flows, and performs precise fault injection. Nagaraj *et al.* [144] proposed to diagnose the performance failures by comparing the logs of system behaviors in good and bad performance. Based on the two collected sets of logs, they applied a number of machine learning techniques to automatically compare and infer the strongest associations between performance and system components, where failures were likely to happen.

Similarly, CAM [82] applies the same idea to the cause analysis for test alarms in system and integration testing. In detail, the failure matching is achieved by using the K nearest neighbors (KNN) algorithm to find similar attribute vectors, which are built on test log terms extracted by term frequency–inverse document frequency (TF-IDF). Furthermore, Amar et al [2] extended the CAM by keeping failing logs and removing logs that passed the test. Then, the most relevant logs in historical failures were extracted by a modified TF-IDF and then vectorized. The vectors were utilized to train an exclusive version of KNN to identify possible log lines that led to the failure. Yuan *et al.* [191] facilitated the failure diagnosis by leveraging historical failures. They built four classifiers by vectorizing historical failures logs using natural language processing techniques. When a new failure occurs, the corresponding classifier was employed to identify the root cause. Moreover, some failures are indistinguishable because they generate very similar logs. To tackle the problem, Ikeuchi et al. [79] proposed to utilize user actions. At first, a log-based failure database is constructed by performing various user actions. In the online deployment phase, when a user action is performed, operators match the failures with the database to identify the root cause.

6.6 Others

There are other valuable topics to broaden the log mining literature. In this section, we mainly introduce two reliability engineering tasks: specification mining and log-based software testing.

Specification Mining. Specification mining is the task of extracting specifications (e.g., program invariants) from program execution traces. Typical specifications impose constraints on the sequencing of program executions (ordering constraint) and program values (value constraint) [114]. Such specifications play an important role in system comprehension, verification, and evolution. Ernst *et al.* [50, 51] leveraged dynamic techniques to discover program invariants from execution traces. Specifically, they instrumented programs to write variables at critical program points (*e.g.*, procedure entries and exits) and checked them against a series of potential invariant rules. They also released an open-source toolkit named Daikon [36] for public reuse, which supports dynamic detection of likely invariants from program computes. Lo *et al.* [114] mined modal scenario-based specifications in the framework of Damm and Harel's Live Sequence Charts (LSC) [37] from execution traces, which describe the relations among caller/callee object identifier, and method signature. Lo *et al.* [112] further extended this model to explore the trigger-and-effect relationship of different charts in LSCs. Moreover, they found the combination of scenario-based and value-based specification mining is able to yield more rich specifications [113]. Using the ITU-T standard Use Case Maps language, Hassine *et al.* [69] visualized execution traces to recover availability scenarios. Such availability scenarios describe the degree to which a system or a component is operational and accessible. In modern user-intensive applications, understanding user behaviors is crucial for their design. To this end, Ghezzi et al. [60] inferred a set of Markov models from user interaction logs to capture their behaviors probabilistically. Lemieux *et al.* [96] introduced a tool named Texada to discover temporal behaviors of programs that hold true on log traces.

Log-based Software Testing. Log-based software testing aims to stimulate the overall software workflow under testing by the generated log data. Many existing work solves this problem by proposing a variety of state-machine-based formalism, similar to the techniques introduced in Section 6.2. A pioneer work [4] discussed the availability of using formal log analyzer to enhance software reliability. The majority of log-based testing approaches utilize cues from log files to improve testing coverage or completeness, Chen *et al.* [21] designed a framework called LogCoCo to estimate code coverage measures using the readily available execution logs. Chen *et al.* [22] derived the state machine framework to conduct workload testing by extracting representative user behaviors from large-scale systems. Cohen et al. [33] proposed a concept called log confidence to evaluate the completeness of any log-based dynamic specification mining task.

7 OPEN-SOURCE TOOLKITS AND DATASETS

In recent years, there are many research studies focusing on automated log analysis. Meanwhile, a number of log management tools have been developed to facilitate the real-world deployment in industry. In this section, we review existing open-source tools and datasets for automated log analysis. To select the desired tools, we define the following criteria: First, we did not consider commercial log analysis tools, such as Splunk [168] and Loggly [119]; Second, the selected tools should cover multiple log analysis phases presented in the paper. After carefully searching, we curated a list of open-source log analysis toolkits, as shown in Table 5. These log analysis tools are GrayLog, GoAccess, Fluentd, Logstash, Logalyze, Prometheus, Syslog-ng, and LogPAI. In common, these tools support crucial features such as log collection, searching, routing, parsing, visualization, alerting and automated analysis. Among these tools, LogPAI is an open-source project built upon pure research outputs while other tools are packed as open-source enterprise products. Additionally, most of existing tools focus on phases before the log mining (*e.g.*, log parsing), and the resulting outcomes (*e.g.*, visualization) are presented to developers for manual analysis. Particularly, LogPAI provides a set of automated log analysis tools such as log parsing and anomaly detection.

Table 5. A list of open-source log analysis tools and log datasets.

	Category	Description	Link
Tools	GrayLog	A centralized log collection and real-time analysis tool.	[63]
	GoAccess	A real-time terminal-based log analyzer for web logs.	[61]
	Fluentd	A data collector that unifies multiple logging infrastructures.	[56]
	Logstash	A server-side processor that ingests and transforms log data.	[122]
	Logalyze	A centralized log management and network monitoring software.	[117]
	Prometheus	A monitoring solution for metrics collection and alerting.	[152]
	Syslog-ng	A scalable log collector and processor across infrastructures.	[169]
	LogPAI	A comprehensive automated log analysis toolkit	[121]
Datasets	Loghub	A large collection of log datasets from various systems.	[120]
	Failure dataset	Error logs produced by OpenStack cloud system.	[40]
	CFDR	Computer failure data repository of different systems.	[15]
	SecRepo	A list of security log data such as malware and system logs.	[159]
	EDGAR	Apache log files that record and store user access statistic.	[47]

Besides the open-source toolkits, a number of log datasets collected from various software systems are available for research studies in academia. Table 5 shows the list of public log datasets and their descriptions. The Loghub [77] maintains a large log collection that is freely accessible for research purposes. Some log datasets are production data released from previous studies, while others are collected in the lab environment. Another research study [34] presented error logs produced by injecting failures in the OpenStack cloud management system. The computer failure data repository (CFDR) focuses mainly on component failure logs in a variety of large production systems. Besides these log datasets, there are also logs collected for other purposes, such as security and web search. SecRepo is a curated list of security data such as malware and system logs. The EDGAR dataset contains Apache logs that record user access statistic through internet search.

8 CONCLUSION

Recent years have witnessed the blossom of log analysis. Given the importance of software logs in reliability engineering, extensive efforts have been contributed to efficient and effective log analysis. This survey mainly explores the four main steps in automated log analysis framework: logging, log compression, log parsing, and log mining. Additionally, we introduce the available open-source toolkits and datasets. Our article enables the outsiders to step in this promising and practical field, and allows the experts to fill in the gaps of their knowledge background. Based on the investigation of these recent advances, we proposed new insights and discussed several future directions, including how to make automated log analysis more feasible under the modern agile and distributed development style, and the concept of a next-generation log analysis framework.

ACKNOWLEDGMENTS

The authors appreciate informative and insightful comments provided by anonymous reviewers, which significantly improve the quality of this survey.

REFERENCES

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 74–89.

[2] Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and localize faults in the test logs. In *Proc. of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 140–151.

[3] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *Proc. of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 49–59.

[4] James H. Andrews. 1998. Testing using log file analysis: tools, methods, and issues. In *Proc. of the thirteenth IEEE conference on automated software engineering ASE*. 157.

[5] AspectJ. 2020. Eclipse AspectJ. Retrieved September 1, 2020 from https://www.eclipse.org/aspectj/

[6] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. 2009. AVA: automated interpretation of dynamically detected anomalies. In *Proc. of the eighteenth international symposium on Software testing and analysis (ISSTA)*. 237–248.

[7] Raju Balakrishnan and Ramendra K Sahoo. 2006. Lossless compression for large scale cluster logs. In *Proc. 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 7.

[8] Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2019. Statistical log differencing. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 851–862.

[9] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The bones of the system: A case study of logging and telemetry at microsoft. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 92–101.

[10] Eduardo Berrocal, Li Yu, Sean Wallace, Michael E. Papka, and Zhiling Lan. 2014. Exploring void search for fault detection on extreme scale systems. In *Proc. of the IEEE international conference on cluster computing (CLUSTER)*. 1–9.

[11] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proc. of the IEEE/ACM 36th International Conference on Software Engineering (ICSE)*. 468–479.

[12] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. of the 19th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 267–277.

[13] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers* 100, 6 (1972), 592–597.

[14] Nimrod Busany and Shahar Maoz. 2016. Behavioral log analysis with statistical guarantees. In *Proc. of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 877–887.

[15] CFDR. 2020. Computer failure data repository. Retrieved September 1, 2020 from https://www.usenix.org/cfdr

[16] An Ran Chen. 2019. An empirical study on leveraging logs for debugging production failures. In *Proc. of the IEEE/ACM 41st International Conference on Software Engineering: Companion (ICSE-Companion)*. 126–128.

[17] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. 71–81.

[18] Boyuan Chen and Zhen Ming Jack Jiang. 2017. Characterizing logging practices in Java-based open source software projects–a replication study in Apache Software Foundation. *Empirical Software Engineering* 22, 1 (2017), 330–374.

[19] Boyuan Chen and Zhen Ming Jack Jiang. 2019. Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering* 24, 4 (2019), 2285–2322.

[20] Boyuan Chen and Zhen Ming (Jack) Jiang. 2020. Studying the use of java logging utilities in the wild. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 397–408.

[21] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 305–316.

[22] Jinfu Chen, Weiyi Shang, Ahmed E. Hassan, Yong Wang, and Jiangbin Lin. 2019. An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour. In *Proc. of the 34th IEEE/ACM international conference on automated software engineering (ASE)*. 669–681.

[23] Kai Chen, Stephen R Schach, Liguo Yu, Jeff Offutt, and Gillian Z Heller. 2004. Open-source change logs. *Empirical Software Engineering* 9, 3 (2004), 197–210.

[24] Rui Chen, Shenglin Zhang, Dongwen Li, Yuzhe Zhang, Fangrui Guo, Weibin Meng, Dan Pei, Yuzhi Zhang, Xu Chen, and Yuqing Liu. 2020. Cross-system log anomaly detection for software systems. In *Proc. of the 31th International Symposium on Software Reliability Engineering (ISSRE)*. 37–47.

[25] Yujun Chen, Yong Xu, Hao Li, Yu Kang, Xian Yang, Qingwei Lin, Hongyu Zhang, Feng Gao, Zhangwei Xu, Yingnong Dang, Dongmei Zhang, and Hang Dong. 2019. Outage prediction and diagnosis for cloud service systems. In *Proc. of the World Wide Web Conference (WWW)*. 2659–2665.

[26] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jack Jiang. 2018. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering* 23, 3 (2018), 1422–1456.

[27] Robert Christensen and Feifei Li. 2013. Adaptive log compression for massive log data. In *Proc. of the SIGMOD Conference*. 1283–1284.
[28] Edward Chuah, Arshad Jhumka, James C Browne, Bill Barth, and Sai Narasimhamurthy. 2015. Insights into the diagnosis of system failures from cluster message logs. In *Proc. of European Dependable Computing Conference (EDCC)*. 225–232.
[29] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. 2010. Diagnosing the root-causes of failures from cluster log files. In *Proc. of the International Conference on High Performance Computing (HPC)*. 1–10.
[30] Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, and Antonio Pecchia. 2014. What logs should you look at when an application fails? insights from an industrial case study. In *Proc. of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 690–695.
[31] Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Antonio Pecchia. 2010. Assessing and improving the effectiveness of logs for the analysis of software faults. In *Proc. of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 457–466.
[32] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2012. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering* 39, 6 (2012), 806–821.
[33] Hila Cohen and Shahar Maoz. 2015. Have we seen enough traces?. In *Proc. of the 30th IEEE/ACM international conference on automated software engineering (ASE)*. 93–103.
[34] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. 2019. How bad can a bug get? an empirical analysis of software failures in the OpenStack cloud computing platform. In *Proc. of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 200–211.
[35] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).
[36] Daikon. 2020. A dynamic invariant detector. Retrieved September 1, 2020 from http://plse.cs.washington.edu/daikon/
[37] Werner Damm and David Harel. 2001. LSCs: Breathing life into message sequence charts. *Formal methods in system design* 19, 1 (2001), 45–80.
[38] Anwesha Das, Frank Mueller, and Barry Rountree. 2020. Aarohi: making real-time node failure prediction feasible. In *Proc. pf the IEEE international parallel and distributed processing symposium (IPDPS)*. 1092–1101.
[39] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in HPC. In *Proc. of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 40–51.
[40] Failure dataset. 2020. Error logs produced by OpenStack. Retrieved September 1, 2020 from https://figshare.com/articles/Failure_dataset/7732268/2
[41] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.
[42] Sebastian Deorowicz and Szymon Grabowski. 2008. Sub-atomic field processing for improved web log compression. In *Proc. of the IEEE International Conference on "Modern Problems of Radio Engineering, Telecommunications and Computer Science" (TCSET)*. 551–556.
[43] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Mining historical issue repositories to heal large-scale online service systems. In *Proc. of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 311–322.
[44] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proc. of the 2015 USENIX Annual Technical Conference (ATC)*. 139–150.
[45] Min Du and Feifei Li. 2018. Spell: Online streaming parsing of large unstructured system logs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2213–2227.
[46] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: anomaly detection and diagnosis from system logs through deep learning. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1285–1298.
[47] EDGAR. 2020. Apache log files. Retrieved September 1, 2020 from https://www.sec.gov/dera/data/edgar-log-file-data-set.html
[48] Nosayba El-Sayed and Bianca Schroeder. 2013. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Proc. of the 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. 1–12.
[49] ELK. 2012. ElasticSearch. Retrieved September 1, 2020 from https://www.elastic.co/elk-stack

[50] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* (2001), 99–123.
[51] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. [n.d.]. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* ([n. d.]), 35–45.
[52] Facebook. 2019. Downtime, outages and failures - understanding their true costs. http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html
[53] Facebook. 2019. Facebook loses $24,420 a minute during outages. https://www.theatlantic.com/technology/archive/2014/10/facebook-is-losing-24420-per-minute/382054/
[54] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. 2015. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *Proc. of the 26th International Symposium on Software Reliability Engineering (ISSRE)*. 24–34.
[55] Bo Feng, Chentao Wu, and Jie Li. 2016. MLC: an efficient multi-level log compression method for cloud backup systems. In *Proc. of the 2016 IEEE Trustcom/BigDataSE/ISPA*. 1358–1365.
[56] Fluentd. 2020. An open source data collector for unified logging layer. Retrieved September 1, 2020 from https://www.fluentd.org
[57] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *Proc. of International Conference on Data Mining (ICDM)*. 149–158.
[58] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Proc. of the 36th International Conference on Software Engineering (ICSE)*. 24–33.
[59] Xiaoyu Fu, Rui Ren, Sally A McKee, Jianfeng Zhan, and Ninghui Sun. 2014. Digging deeper into cluster system logs for failure prediction and root cause diagnosis. In *Proc. of International Conference on Cluster Computing (CLUSTER)*. 103–112.
[60] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*. 277–287.
[61] GoAccess. 2020. A fast, terminal-based log analyzer. Retrieved September 1, 2020 from https://goaccess.io
[62] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience report: Log-based behavioral differencing. In *Proc. of the 28th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 282–293.
[63] GrayLog. 2020. A leading centralized log management solution. Retrieved September 1, 2020 from https://www.graylog.org
[64] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: fast pattern recognition for log analytics. In *Proc. of the 25th ACM international conference on Information and knowledge management (CIKM)*. 1573–1582.
[65] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. 2006. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC)*. 181–190.
[66] Stephen E Hansen and E Todd Atkins. 1993. Automated system monitoring and notification with swatch.. In *Proc. of the 7th USENIX Large Installation System Administration Conference (LISA)*, Vol. 93. 145–152.
[67] Ahmed E Hassan, Daryl J Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. 2008. An industrial case study of customizing operational profiles using log compression. In *Proc. of the 30th international conference on software engineering (ICSE)*. 713–723.
[68] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering* (2018), 3248–3280.
[69] Jameleddine Hassine, Abdelwahab Hamou-Lhadj, and Luay Alawneh. 2018. A framework for the recovery and visualization of system availability scenarios from execution traces. *Information and Software Technology* (2018), 78–93.
[70] Kimmo Hätönen, Jean François Boulicaut, Mika Klemettinen, Markus Miettinen, and Cyrille Masson. 2003. Comprehensive log compression with frequent patterns. In *Proc. of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*. Springer, 360–370.
[71] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the natural language descriptions in software logging statements. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 178–189.
[72] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2017. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2017), 931–944.
[73] Pinjia He, Jieming Zhu, Pengcheng Xu, Zibin Zheng, and Michael R. Lyu. 2018. A directed acyclic graph approach to online log parsing. *arXiv preprint arXiv:1806.04356* (2018).

[74] Pinjia. He, Jieming. Zhu, Zibin. Zheng, and Michael R. Lyu. 2017. Drain: an online log parsing approach with fixed depth tree. In *Proc. of the 24th International Conference on Web Services (ICWS)*. 33–40.
[75] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proc. of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 60–70.
[76] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2016. Experience report: System log analysis for anomaly detection. In *Proc. of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 207–218.
[77] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *arXiv preprint arXiv:2008.06448* (2020).
[78] Toan Huynh and James Miller. 2009. Another viewpoint on "evaluating web software reliability based on workload and failure data extracted from server logs". *Empirical Software Engineering* (2009), 371–396.
[79] Hiroki Ikeuchi, Akio Watanabe, Takehiro Kawata, and Ryoichi Kawahara. 2018. Root-cause diagnosis using logs generated by user actions. In *Proc. of the IEEE Global Communications Conference (GLOBECOM)*. 1–7.
[80] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *Proc. of the IEEE International Conference on Web Services (ICWS)*. 25–32.
[81] Tong Jia, Lin Yang, Pengfei Chen, Ying Li, Fanjing Meng, and Jingmin Xu. 2017. Logsed: anomaly diagnosis through mining time-weighted control flow graph in logs. In *Proc. of the IEEE 10th International Conference on Cloud Computing (CLOUD)*. 447–455.
[82] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In *Proc. of the IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 712–723.
[83] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. 2009. Understanding customer problem troubleshooting from storage system logs. In *Proc. of the 7th conference on File and storage technologies (FAST)*. 43–56.
[84] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications. In *Proc. of the Eighth International Conference on Quality Software (QSIC)*.
[85] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *Proc. of the 24th IEEE International Conference on Software Maintenance (ICSM)*. 307–316.
[86] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging library migrations: a case study for the apache software foundation projects. In *Proc. of the 13th International Conference on Mining Software Repositories (MSR)*. 154–164.
[87] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. 2018. Examining the stability of logging statements. *Empirical Software Engineering* (2018), 290–333.
[88] Soila P Kavulya, Kaustubh Joshi, Felicita Di Giandomenico, and Priya Narasimhan. 2012. Failure diagnosis of complex systems. In *Resilience assessment and evaluation of computing systems*. Springer, 239–261.
[89] Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Mustapha Aminu Bagiwa, Muhammad Shiraz, Samee U Khan, Rajkumar Buyya, and Albert Y Zomaya. 2016. Cloud log forensics: Foundations, state of the art, and future directions. *ACM Computing Surveys (CSUR)* (2016), 178–184.
[90] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer.
[91] Tatsuaki Kimura, Akio Watanabe, Tsuyoshi Toyono, and Keisuke Ishibashi. 2018. Proactive failure detection learning generation patterns of large-scale network logs. *IEICE Transactions on Communications* (2018).
[92] Jason King, Jon Stallings, Maria Riaz, and Laurie Williams. 2017. To log, or not to log: using heuristics to identify mandatory log events–a controlled experiment. *Empirical Software Engineering* (2017), 2684–2717.
[93] Jannis Klinkenberg, Christian Terboven, Stefan Lankes, and Matthias S. Müller. 2017. Data mining-based analysis of hpc center operations. In *Proc. of the IEEE International Conference on Cluster Computing (CLUSTER)*. 766–773.
[94] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. 2020. System log clustering approaches for cyber security applications: A survey. *Computers & Security* 92 (2020), 101739.
[95] Jean-Claude Laprie. 1995. Dependable computing: Concepts, limits, challenges. In *Special issue of the 25th international symposium on fault-tolerant computing (FTCS)*. 42–54.
[96] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 81–92.
[97] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. *Empir. Softw. Eng.* (2018), 2655–2694.
[98] Heng Li, Weiyi Shang, and Ahmed E Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* (2017), 1684–1716.

[99] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering* (2017), 1831–1865.
[100] Man-Lap Li, Pradeep Ramachandran, Swarup K Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. 2008. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN)*. 22–31.
[101] Shanshan Li, Xu Niu, Zhouyang Jia, Xiangke Liao, Ji Wang, and Tao Li. 2020. Guiding log revisions by learning from software evolution history. *Empir. Softw. Eng.* (2020), 2302–2340.
[102] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *Proc. of the 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[103] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. Dlfinder: characterizing and detecting duplicate logging code smells. In *Proc. of the 41st International Conference on Software Engineering (ICSE)*. 152–163.
[104] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Proc. of the 7th IEEE International Conference on Data Mining (ICDM)*. 583–588.
[105] Chinghway Lim, Navjot Singh, and Shalini Yajnik. 2008. A log mining approach to failure analysis of enterprise telephony systems. In *Proc. of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[106] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. 2015. Cowic: A column-wise independent compression for log stream analysis. In *Proc. of the 15th International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*.
[107] Qingwei Lin, Randolph Yao, Murali Chintalapati, Dongmei Zhang, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, and Youjiang Wu. 2018. Predicting node failure in cloud service systems. In *Proc. of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 480–490.
[108] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proc. of the 38th International Conference on Software Engineering Companion (ICSE-Companion)*. 102–111.
[109] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. 2019. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Proc. of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1777–1794.
[110] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. 2019. Logzip: extracting hidden structures via iterative clustering for log compression. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 863–873.
[111] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which variables should I log? *IEEE Transactions on Software Engineering* (2019).
[112] David Lo and Shahar Maoz. 2008. Mining scenario-based triggers and effects. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 109–118.
[113] David Lo and Shahar Maoz. 2012. Scenario-based and value-based specification mining: better together. *Automated Software Engineering* (2012), 423–458.
[114] David Lo, Shahar Maoz, and Siau-Cheng Khoo. 2007. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 465–468.
[115] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The fuzzylog: a partially ordered shared log. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 357–372.
[116] Log4j. 2020. Apache Log4j. Retrieved September 1, 2020 from http://logging.apache.org/log4j/
[117] Logalyze. 2020. An open source log management and network monitoring software. Retrieved September 1, 2020 from http://www.logalyze.com
[118] Loggly. 2009. Automated parsing log types. Retrieved September 1, 2020 from https://www.loggly.com/docs/automated-parsing
[119] Loggly. 2009. Loggly - log management by loggly. Retrieved September 1, 2020 from https://www.loggly.com
[120] Loghub. 2020. A large collection of log datasets from various systems. Retrieved September 1, 2020 from https://github.com/logpai/loghub/
[121] LogPAI. 2020. A platform for log analytics powered by AI. Retrieved September 1, 2020 from https://www.logpai.com
[122] Logstash. 2020. A server-side processor for log data. Retrieved September 1, 2020 from https://www.elastic.co/logstash
[123] logz. 2014. Log parsing - automated, easy to use, and efficient. Retrieved September 1, 2020 from https://logz.io/product/log-parsing
[124] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. 2010. Mining program workflow from interleaved traces. In *Proc. of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (SIGKDD)*.

[125] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection.. In *Proc. of the 2010 USENIX Annual Technical Conference (ATC)*. 1–14.
[126] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proc. of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 3–14.
[127] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. 2017. Log-based abnormal task detection and root cause analysis for spark. In *Proc. of the IEEE International Conference on Web Services (ICWS)*.
[128] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. 2018. Detecting anomaly in big data system logs using convolutional neural network. In *Proc. of the 16th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*. 151–158.
[129] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. 2018. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*. 321–334.
[130] Michael R. Lyu (ed.). 1996. *Handbook of software reliability engineering*. IEEE Computer Society Press.
[131] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2012. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2012).
[132] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proc. of International Conference on Knowledge Discovery and Data Mining (KDD)*. 1255–1264.
[133] Michael Meinig, Peter Tröger, and Christoph Meinel. 2019. Rough logs: a data reduction approach for log files. In *Proc. of the International Conference on Enterprise Information Systems (ICEIS)*. 295–302.
[134] Peter Mell and Richard E Harang. 2014. Lightweight packing of log files for improved compression in mobile tactical networks. In *Proc. of the IEEE Military Communications Conference (MILCOM)*. 192–197.
[135] Weibin Meng, Ying Liu, Yuheng Huang, Shenglin Zhang, Federico Zaiter, Bingjin Chen, and Dan Pei. 2020. A semantic-aware representation framework for online log analysis. (2020), 1–7.
[136] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *Proc. of the 2019 International Joint Conferences on Artificial Intelligence (IJCAI)*. 4739–4745.
[137] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *Proc. of the IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 167–16710.
[138] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael R. Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.
[139] Microsoft. 2018. Event logging. Retrieved September 1, 2020 from https://docs.microsoft.com/en-us/windows/win32/eventlog/event-logging
[140] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proc. of the 27th Conference on Neural Information Processing Systems (NIPS)*. 3111–3119.
[141] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. 2019. PADLA: a dynamic log level adapter using online phase detection. In *Proc. of the 27th International Conference on Program Comprehension (ICPC)*. 135–138.
[142] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *Proc. of the IEEE International Conference on Services Computing (SCC)*. 595–602.
[143] Meiyappan Nagappan and Mladen A Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*. 114–117.
[144] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 353–366.
[145] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proc. of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 215–224.
[146] Adam J. Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *ACM Communication* (2012), 55–61.
[147] Frederick John Otten et al. 2008. *Using semantic knowledge to improve compression on log files*. Ph.D. Dissertation. Rhodes University.
[148] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Network and Distributed System Security Symposium*.

[149] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: assessment of a critical software development process. In *Proc. of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. 169–178.
[150] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2016. Failure diagnosis for distributed systems using targeted fault injection. *Transactions on Parallel and Distributed Systems* 28, 2 (2016), 503–516.
[151] James E Prewett. 2003. Analyzing cluster log files using logsurfer. In *Proc. of the 4th Annual Conference on Linux Clusters*. Citeseer.
[152] Prometheus. 2020. A systems and service monitoring system. Retrieved September 1, 2020 from https://github.com/prometheus
[153] Balázs Rácz and András Lukács. 2004. High density compression of log files. In *Proc. of the Data Compression Conference (DCC)*. IEEE, 557–557.
[154] Rapid7. 2000. New automated log parsing. Retrieved September 1, 2020 from https://blog.rapid7.com/2016/03/03/new-automated-log-parsing
[155] John P Rouillard. 2004. Real-time log file analysis using the simple event correlator (sec).. In *Proc. of the 18th USENIX Large Installation System Administration Conference (LISA)*, Vol. 4. 133–150.
[156] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. 2015. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering* 20, 4 (2015), 879–927.
[157] Ramendra K. Sahoo, Adam J. Oliner, Irina Rish, Manish Gupta, José E. Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. 2003. Critical event prediction for proactive management in large-scale computer clusters. In *Proc. of the ninth ACM SIGKDD international conference on knowledge discovery and data mining (SIGKDD)*. 426–435.
[158] Bianca Schroeder and Garth A. Gibson. 2006. A large-scale study of failures in high-performance computing systems. In *Proc. of the 2006 International Conference on Dependable Systems and Networks (DSN)*. 249–258.
[159] SecRepo. 2020. A list of security log data. Retrieved September 1, 2020 from http://www.secrepo.com
[160] Weiyi Shang. 2012. Bridging the divide between software developers and operators using logs. In *Proc. of the 34th International Conference on Software Engineering (ICSE)*. 1583–1586.
[161] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proc. of the 35th International Conference on Software Engineering (ICSE)*. 402–411.
[162] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering* (2015), 1–27.
[163] Weiyi Shang, Meiyappan Nagappan, Ahmed E. Hassan, and Zhen Ming Jiang. 2014. Understanding log lines using development knowledge. In *Proc. of the 30th International Conference on Software Maintenance and Evolution (ICSME)*.
[164] Keiichi Shima. 2016. Length matters: clustering system log messages using length of words. *arXiv:1611.03213* (2016).
[165] Przemysław Skibiński and Jakub Swacha. 2007. Fast and efficient log file compression. In *Proc. of 11th east-European conference on advances in databases and information systems (ADBIS)*. 56–69.
[166] SLF4J. 2020. Simple Logging Facade for Java (SLF4J). Retrieved September 1, 2020 from http://www.slf4j.org/
[167] spdlog. 2020. Spdlog. Retrieved September 1, 2020 from https://github.com/gabime/spdlog
[168] Splunk. 2005. Splunk platform. Retrieved September 1, 2020 from http://www.splunk.com
[169] Syslog-ng. 2020. A log management solution. Retrieved September 1, 2020 from https://www.syslog-ng.com
[170] Byung Chul Tak, Shu Tao, Lin Yang, Chao Zhu, and Yaoping Ruan. 2016. Logan: problem diagnosis in the cloud using log-based reference models. In *Proc. of the IEEE International Conference on Cloud Engineering (IC2E)*. 62–67.
[171] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proc. of the 20th ACM international conference on Information and knowledge management (CIKM)*. 785–794.
[172] StatusCake Team. 2020. The Most Expensive Website Downtime Periods in History. https://www.statuscake.com/the-most-expensive-website-downtime-periods-in-history/
[173] UpGuard. 2019. The cost of downtime at the world's biggest online retailer. Retrieved September 1, 2020 from https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer
[174] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proc. of the 3rd IEEE Workshop on IP Operations & Management (IPOM)*. 119–126.
[175] Risto Vaarandi and Mauno Pihelgas. 2015. LogCluster - A data clustering and pattern mining algorithm for event logs. In *Proc. of the 11th International conference on network and service management (CNSM)*. 1–7.
[176] Wil Van Der Aalst. 2012. Process mining. *Commun. ACM* (2012), 76–83.
[177] Bin Xia, Yuxuan Bai, Junjie Yin, Yun Li, and Jian Xu. 2020. LogGAN: a log-level generative adversarial network for anomaly detection using permutation event modeling. *Information Systems Frontiers* (2020), 1–14.
[178] Wei Xu. 2010. *System problem detection by mining console logs*. Ph.D. Dissertation. University of California, Berkeley.

[179] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online system problem detection by mining patterns of console logs. In *Proc. of the Ninth IEEE International Conference on Data Mining (ICDM)*. 588–597.

[180] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*.

[181] Kenji Yamanishi and Yuko Maruyama. 2005. Dynamic syslog mining for network failure monitoring. In *Proc. of the 11nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 499–508.

[182] Stephen Yang, Seo Jin Park, and John K. Ousterhout. 2018. NanoLog: A nanosecond scale logging system. In *Proc. of the 2018 USENIX Annual Technical Conference (ATC)*. 335–350.

[183] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empir. Softw. Eng.* (2020), 488–531.

[184] Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E Hassan. 2019. A study of the performance of general compressors on log files. *Empirical Software Engineering* (2019), 3043–3085.

[185] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending performance from real-world execution traces: A device-driver case. In *Proc. of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*. 193–206.

[186] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: workflow monitoring of cloud infrastructures via interleaved logs. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 489–502.

[187] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proc. of the fifteenth International Conference on Architectural support for programming languages and operating systems (ASPLOS)*. 143–154.

[188] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 293–306.

[189] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proc. of 34th International Conference on Software Engineering (ICSE)*. 102–112.

[190] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 3–14.

[191] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. 2019. An approach to cloud execution failure diagnosis based on exception logs in openstack. In *Proc. of the IEEE 12th International Conference on Cloud Computing (CLOUD)*. 124–131.

[192] Tarannum Shaila Zaman, Xue Han, and Tingting Yu. 2019. SCMiner: localizing system-level concurrency faults from large system call traces. In *Proc. of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[193] Lei Zeng, Yang Xiao, Hui Chen, Bo Sun, and Wenlin Han. 2016. Computer operating system logging and security issues: a survey. *Security and Communication Networks* 9, 17 (2016), 4804–4821.

[194] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering* (2019), 3394–3434.

[195] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, Yu Chen, Hui Dong, Xianping Qu, and Lei Song. 2018. PreFix: switch failure prediction in datacenter networks. *Proc. ACM Meas. Anal. Comput. Syst.* 2 (2018), 2:1–2:29.

[196] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proc. of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[197] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proc. of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 131–146.

[198] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. The game of twenty questions: do you know where to log?. In *Proc. of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*.

[199] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: fully automated optimal placement of log printing statements under specified overhead threshold. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*. 565–581.

[200] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proc. of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. 603–618.

[201] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software*

Engineering (2018).

[202] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proc. of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 683–694.

[203] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: helping developers make informed logging decisions. In *Proc. of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. 415–425.

[204] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and benchmarks for automated log parsing. In *Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 121–130.

[205] Yuan Zuo, Yulei Wu, Geyong Min, Chengqiang Huang, and Ke Pei. 2020. An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis. *IEEE Transactions on Cognitive Communications and Networking* (2020).