electronics

MDPI

Review

Software Failure Log Analysis for Engineers—Review

Wojciech Dobrowolski [1,2,*], Maciej Nikodem [2] and Olgierd Unold [2]

[1] Nokia, Rodziny Hiszpanskich 8, 02-685 Warszawa, Poland
[2] Politechnika Wroclawska, Wybrzeze Stanisława Wyspianskiego 27, 50-370 Wrocław, Poland; maciej.nikodem@pwr.edu.pl (M.N.); olgierd.unold@pwr.edu.pl (O.U.)
* Correspondence: wojciech.dobrowolski@nokia.com

Abstract: The use of automated methods for log analysis is unavoidable in any large company; therefore, it has attracted attention from engineers and researchers. As a result, the number of articles in the field grows yearly and new approaches are frequently proposed. Unfortunately, published research works only sometimes meet the needs of engineers wishing to apply the methods in real-life systems. A common issue is that the method's benefits often do not compensate for the effort required for its implementation and maintenance. Therefore, engineers must understand the pros and cons of full-scale applications, including the implementation details and the required effort. This work provides a comprehensive review of automated log analysis methods and aims to offer a guide for software engineers who fix integration and production failures. The article categorizes and provides an overview of existing methods and assesses their implementation and maintenance costs, as well as the feasibility of the methods. The article also identifies and describes the shortcomings of existing methods, including concept drift, which is not addressed with sufficient attention, as well as the lack of online benchmarks and the interpretation of the log sequence as a language, without an in-depth analysis of its properties. Despite growing efforts to provide feasible and widely adopted solutions, many reference implementations are unavailable. Consequently, the time and computation complexities differ between various implementations of the same approaches, making the results of research work difficult to replicate in real-life scenarios.

Keywords: software engineering; software failures; log mining; log analysis; implementation; engineer

Citation: Dobrowolski, W.; Nikodem, M.; Unold, O. Software Failure Log Analysis for Engineers—Review. *Electronics* **2023**, *12*, 2260. https://doi.org/10.3390/electronics12102260

Academic Editors: Seonah Lee, Jinhyun Kim, Suwon Lee and Iouliia Skliarova

Received: 3 April 2023
Revised: 13 May 2023
Accepted: 15 May 2023
Published: 16 May 2023

1. Introduction

When an engineer wishes to apply a scientific principle, they need precise information on the available methods and who can use them. At this stage of the software lifetime, their performance, requirements, and costs of implementation are crucial. This point of view needs to be improved in the automated log analysis field.

Logs are the most common source of information available for analysis when a system failure occurs. In the world of growing software systems, logs have thousands of lines. A failure can be indicated by a fatal, error, or even info log message, which adds another level of complexity [1]. Consequently, the manual extraction of information from logs is infeasible for human operators, and automatic methods are needed. Unfortunately, universal and scalable methods still need to be developed or have too many limits. For example, static methods are quickly outdated, are prone to noise, and have a large computation overhead. Machine learning (ML) and deep learning (DL) methods can overcome these difficulties. Unfortunately, a universal ML or DL method has yet to be proposed, and engineers need to search across different approaches and solutions.

One reason for the lack of universal methods is the nature of system failures, which may have various causes. Runtime errors may be due to a software fault or bug, the incorrect configuration of software components, or missing functionality. The reason for failure and the time at which the failures are encountered may differ. Some errors are detected during tests in the software development stage, while others go undetected in the

testing procedures and manifest themselves in production. Depending on the reasons and the time at which failures are detected, the amount of information available to the software engineer and test engineer changes and the complexity of the task to find the cause of the failure grows.

The information for errors detected in integration tests is usually limited to logs, the source code, and the execution history. The situation is even more difficult for errors detected in production because, in such a case, usually only the execution logs are available for analysis.

Another reason for the need for a universal method is that error-related logs are analyzed by several groups of people in the company, each having different expectations regarding the required information details. The test engineer and manager require information on the high-level root cause of the failure and the responsible group of software engineers who should address the error. The engineer wishes to restrict the log to the most relevant parts and obtain more detailed information on the root cause of the failure, ideally connected to the source code line. Existing methods can provide the engineer with much different information. Methods based on anomaly detection allow the engineer to focus on the parts of the logs that are the most likely to contain the failure. There are also methods allowing them to reconstruct the execution path and find the root cause by identifying faulty components or tasks. They can also give a sequence of API calls to reproduce the failure or generate a unit test with failure reproduction.

These methods require various input data. Most of the time, methods utilize only logs collected during faulty execution. However, some methods also need logs of past executions. The outcome of the methods is often quite generic and unclear to engineers. To obtain more meaningful results, they need further processing—the output data have to be grouped and labeled, a unit test has to be provided, or a history of the same test is needed.

Despite many research efforts, the existing approaches need to be revised to allow the adoption of log analysis tools [2]. The reason might be that engineers' expectations differ from those presented in the literature. The accuracy might need to be improved, and the cost of implementation may be too high. The consensus among embedded software engineers is that existing solutions do not provide information on software components' data flow and interactions. There is a need for a tool to identify meaningful differences between executions. They also report the need for a top-down approach, where a high-level hypothesis is verified by a lower level of analysis [3].

Regarding accuracy, when methods were applied to split data chronologically, a typical real-world scenario achieved lower accuracy than in original papers [4].

Most methods are tested on open-source systems, which may differ significantly from commercial systems. Finally, the cost of implementation is difficult to assess. Not all papers publish their source codes and data; many have implicit assumptions about the data and system. Only some present the pseudo-code and time and memory performance measures.

This article addresses the research question of the applicability of log analysis methods to real-life situations and practical applications without the need for frequent intervention or retraining. The main contributions of the article include

- a broad review of existing methods for automated log analysis focusing on approaches applicable to real-life systems;
- classification of the methods into categories important for software engineers: available implementation, algorithms and approaches used, applicable stage, robustness to concept drift, and their performance, including the F1 score.

We refer the reader to [5] for a systematic review of automated log analysis.

The remaining parts of the article are organized as follows. Section 3 defines software failures and categorizes them according to the moment of detection in the software's lifetime, the information available for analysis, and the localization of errors. In Section 4, anomaly detection methods are described, along with retrieval-based and execution replay

methods. Section 5 summarizes the review and presents the assessment of the reviewed methods, while Section 6 draws conclusions and presents further investigation areas.

2. Review Methodology

We consulted the Mendeley Library to systematically collect and revise publications for this review. We first searched relevant papers in online digital libraries and extended the repository by manually inspecting all references in these papers. To begin with, we explored several popular online digital libraries (e.g., IEEE Xplore, ACM Digital Library, Springer Online, Elsevier Online, and ScienceDirect) with the following keywords: "log + automated analysis", "log + automated analysis + industry", "log + automated analysis + engineer", "log + automated analysis + production", "log + automated analysis + software development". Then, we manually inspected each reference of these papers to collect additional publications related to the survey topics. This article focused on publications studying the reliability issues with software system logs applied in the industry. We took taxonomy from [6] and followed the search in three additional categories: anomaly detection with keywords "log + anomaly detection + production"; retrieval-based methods with keywords "logs + retrieval based + production", "log + failure category+production", "log + failure classification + production"; and execution replay methods with keywords "log + execution replay + production", "log + execution path+production". We collected 43 publications in the automated log analysis area from 2002 to 2022. We found that automated log analysis has been continuously and actively investigated in the past two decades. In particular, we observed (Figure 1) steady growth in the number of publications emphasizing industry implementation since 2018, indicating that the log analysis field has attracted increasing interest since then.

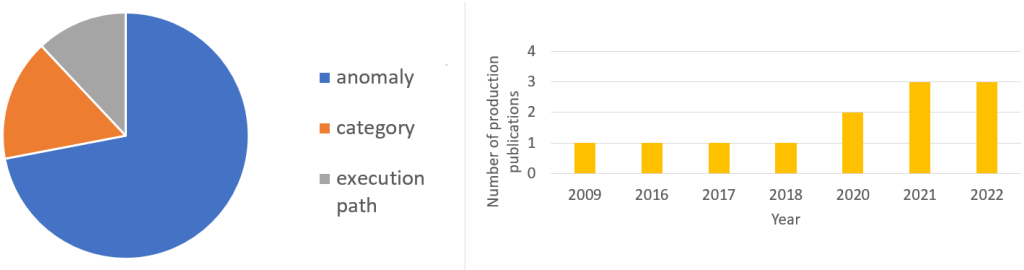

Figure 1. Paper distribution for each topic and the publication trend regarding industrial application of log analysis.

Furthermore, we classified these publications into three categories by research focus: anomaly detection, retrieval-based, and execution replay methods. The reason for this is that, from a practical point of view, the engineer must complete three steps to solve the problem: to find the place where the anomaly originated (anomaly detection), to categorize the failure (failure categorization), and to analyze the execution path (execution replay methods) to find the solution. This is consistent with findings from an interview study of industry developers [3]. The distribution shows that a large portion (18 papers) of the research efforts were devoted to anomaly detection. The reasons are two-fold. First, anomaly detection is the most straightforward binary decision task that can be performed on logs. Second, it can give informative insights into the nature of log sequences and drive more sophisticated machine learning tasks, such as failure categorization. Among the reviewed papers, only 12 mention industrial/practical implementation. Furthermore, we were aware of several existing surveys on automated log analysis. Zhao et al. [2] reviewed 158 log analysis papers and pointed out some challenges in 2022. They analyzed papers in five categories: logging, log compression, log parsing, log mining, and empirical study. Other existing surveys [6–8] focused on log anomaly detection. However, there needs to be more research and implementation. Unlike these studies, our survey mainly targets automated log analysis from a practical point of view.

3. Research Problem

Software failures are deviations in one or more states of the system. This deviation is called an error [9]. The reason for the error is a fault that may be internal or external. Internal faults are software bugs, while external faults include problems with the operating environment, an incorrect configuration, or incorrect input. To deduce the root cause of the error and identify and locate the fault, the engineer needs information on the failed execution of the software. The information might include logs, the history of execution of the test scenarios, the history of commits, debugger and execution traces, input data, and the configuration. The amount of data available to an engineer in a large system is overwhelming.

3.1. Software Lifetime and Fault Diagnosis

Without loss of generality, one can assume that software failures can be detected in one of the three stages of the software lifetime: during unit testing, integration testing [10], or production [11]. Failures encountered in production are the most critical because they are often related to contractual penalties and are associated with high pressure to identify, locate, and solve problems quickly. Production failures relate to the most significant pressure from the company management [11], as they relate to contractual penalties. Unit test failures are on the other end of the spectrum. The engineer often encounters them on the local machine while developing a new feature or regression. Consequently, there is little pressure regarding the time to diagnose and correct the failure. Faults can also be detected during continuous integration tests when part of the functionality is ready. Because functionalities include modifications in many components, it is reasonable to test the behavior of all modifications at once [12]. Consequently, the pressure to solve these issues is relatively high.

The type and amount of data available are different at different stages of the lifetime of the software. Unit testing failures have the richest data available, from the debugging results, through execution traces, to the history of the execution, as shown in Table 1. They are often caused by the latest changes in the current software components, which drastically reduces the search space. They can also be easily rerun with additional flags/additional local logs. This makes them very easy to fix. Automatic methods are limited here, concentrating mostly on spectrum-based failure detection [13,14], which compares executed lines in passing and failing tests to determine which are the most common in failure tests. However, unit tests, even when covering code with a high percentage, cannot verify all possible inputs and paths of execution due to the state explosion problem [15]. It is still visible despite using AI methods to improve unit testing [16,17]. This is why failure in later stages of development is inevitable and must be tackled, especially when modifications in several components, delivering new functionality, are ready, and integration tests are performed to ensure the correctness of the delivered implementation on a multi-component level. The available data are logs, the history of execution, and information on the latest changes. Compared to failures detected in the unit test, the analysis here is more complex, because the cause of the failure may be spread among the different components and parts of the software that have been changed. However, the engineer can rerun the integration tests and analyze the results for a different set of parameters and different configurations to check for a possible root cause and collect additional logs.

The most challenging are failures that occur in production, as they might result from changes made in several components. The failure of one component does not necessarily mean that this component contains the error, as the failure may have occurred elsewhere and is merely observable here. Failure analysis is based mainly on logs containing information on all components' behavior, and additional data collection for debugging purposes are challenging or impossible to obtain. For these reasons, we will focus only on integration tests and production.

Table 1. Failure at different stages with different data available.

Stage	Logs	Source Code	History of Execution	Debugging
Unit Test Failure	✓	✓	✓	✓
Int. Test Failure	✓	✓	✓	
Production Failure	✓	✓		

3.2. Data Available for Error/Failure Analysis

The type and amount of data available are different at different stages of the lifetime of the software. Unit testing failures have the richest data available, from the debugging results, through execution traces, to the history of the execution, as shown in Table 1. They are often caused by the latest changes in the current software components, which drastically reduces the search space. They can also be easily rerun with additional flags/additional local logs. This makes them very easy to fix. Automatic methods are limited here.

When modifications in several components, delivering new functionality, are ready, integration tests are performed to ensure the correctness of the provided implementation on a multi-component level. The available data are the logs, the history of execution, and information on the latest changes. Compared to failures detected in the unit test, the analysis here is more complex, because the cause of the failure may be spread among the different components and parts of the software that have been changed. However, the engineer can rerun the integration tests and analyze the results for different parameters and other configurations to check for a possible root cause and collect additional logs. The engineer performs failure analysis by inspecting the logs using regular expressions or by comparing normal and failed case logs. Studies of the latest changes in the software can be helpful; however, they span many components and can be non-trivial.

The most challenging are failures that occur in production, as they might result from changes made in several components. The failure of one component does not necessarily mean that this component contains the error, as the failure may have occurred elsewhere and is merely observable here. Failure analysis is based mainly on logs containing information on all components' behavior. Additional data collection for debugging purposes is highly difficult or even impossible to obtain. Once again, the engineer usually relies on a manually prepared set of regular expressions and experience in log analysis. The number of logs to analyze is even greater than in the case of integration testing, and the set of the latest changes in the software is also more significant. For these reasons, we will focus only on integration tests and production.

3.3. Industrial Approach to Log Analysis

Nowadays, the fixing of software failures is based on scripts that automate log processing and some analysis steps. Nonetheless, the process significantly depends on the engineers' experience and knowledge. Scripts are relatively simple as they are based on regular expressions or rules that are built individually by each engineer or stored in large rule- based systems. When they fail to explain the error, the only approach left is a tedious search through thousands of lines of logs and source code. Through great effort, the engineer applies his knowledge and experience to detect suspicious instances that may signal the cause of the error. The above approach has two main areas for improvement. First, such scripts, regular expressions, and rules may become outdated as the logs and software constantly change. Second, when an experienced engineer leaves the company, his knowledge is lost to others.

3.4. Scientific Methods

Because the manual examination of thousands of log lines produced by today's systems is not feasible, researchers seek scientific solutions. Log anomaly analysis is growing in popularity in this research area. There are standardized benchmarks available [18], and advanced machine learning and deep learning methods are used [19,20]. A log sequence

is often treated as a natural language [21–24]. Methods can find log lines or segments of lines that are anomalous. The abstraction of a log sequence as a language is also popular. Some researchers go further and classify anomalies [25]. The execution path can also be extracted from the logs [26,27]. One can extract higher- level information on execution, which includes relevant API calls [28]. These methods use an algorithmic approach. Some rely on integration testing [28], while others are applicable on all levels [21]. As a result, the number of publications on the topic grows yearly, but their results are not easily applied to real-life commercial systems [29].

AI Methods

It is worth mentioning that the most successful of the above approaches is the use of AI methods, specifically deep learning [19,30]. It started, however, from rule-based systems such as Logsurfer [31], where the system expert has to input the necessary regular expressions and maintain them, which is a known limitation. Then, the use of machine learning methods allowed the automatic extraction of features, such as the TFIDF method, where logs are firstly parsed into log templates with invariant parts, and the frequency of their appearance in each line and document is used to represent them in the fixed dimension vector. Machine learning methods such as Support Vector Machines, Logistic Regression, and Decision Trees can determine whether feature vectors represent a failure or normal behavior. The limitation here is that the extraction method is fixed, and the model has to be retrained whenever there is a change in input. Ultimately, deep learning methods solve this problem by learning features from log sequences in an NLP manner. This allows them to be more flexible and return sufficient results for previously unseen input. We provide more details in Sections 4.1.1, 4.2.1 and 4.3.1.

3.5. Research Problem

There is a clear gap between research and industry. Methods that achieve good F1 scores are not feasible for implementation because they require frequent retraining. Others present very complex algorithms with no reference implementation, making them challenging and time-consuming to apply in reality. In some cases, available implementations are not complete or lack essential components. It is also challenging for a software engineer to find appropriate methods, as some are suitable for production failures and log analysis, while others can only be applied to integration failures. This article aims to support engineers and address the question of which methods proposed so far are most promising and of practical use, being easy to implement without frequent intervention after training.

4. Approaches to Automated Log Analysis

This section presents different approaches to automated log analysis. We distinguish three main approaches: anomaly detection (Figure 2), retrieval-based methods (Figure 3), and execution replay methods (Figure 4). Each approach is discussed in three parts: first, we discuss the available approaches, followed by how they deal with concept drift, and, finally, the implementation details presented in original papers. Concept drift is a well-known definition [6,8] describing situations whereby log lines are added, deleted, or modified during the software development lifetime. Without proper online learning capability, frequent model retraining is required. The main problem with concept drift is that when the model is trained on historical data that are not sorted by date, it will have access to log patterns from the whole period of time. Meanwhile, in real life, it will have to behave correctly on unseen log patterns. Utilizing a train/test split in this way has the consequence of falsely providing high-precision and -recall results, which was described in [4].

An anomaly is extracted in many proposed approaches, and anomaly detection methods are a fast-growing field of research. The most successful methods make use of ML and DL algorithms. An anomaly can be helpful for the engineer to reduce the number of logs to analyze by selecting a log segment with the anomaly. However, information about the anomaly is more valuable in many contexts, or it can be misleading when looking

for the root cause in a complex system. Researchers make an effort to extract as much information as possible from the logs to ease the work of the manual log analysis. As a consequence, there are methods to extract the execution path or classify the failure category. The execution path is essential for the developer engineer in finding the root cause. At the same time, file categorization helps the test engineer to assign the encountered failure to the correct developer group.

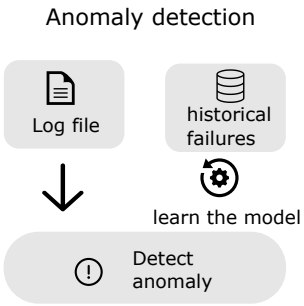

Figure 2. Anomaly detection method consists of learning the model and applying it to detect the anomaly in the log.

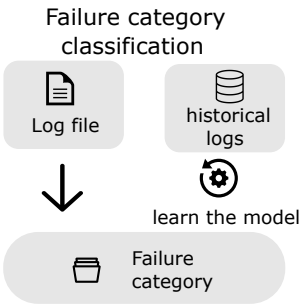

Figure 3. Retrieval-based methods store historical logs along with failure categories and learn a model from this. It is then used to make predictions about new logs.

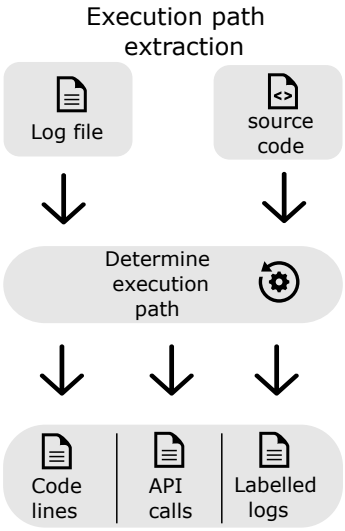

Figure 4. Execution paths process log lines and find the most probable execution path. This might return exact code lines, API calls, or label code segments with the most similar test executions seen during training.

4.1. Anomaly Detection

Systems are becoming increasingly more complex and thus prone to bugs and vulnerabilities. One can detect anomalous behavior from logs. An anomaly is a sequence of

logs or a single log related to a bug or vulnerability. The methods of detecting anomalies consist of log parsing (Figure 5), feature extraction, and classification. The results are anomalous sequences of logs or single logs. Anomaly detection can restrict the number of logs for the developer to analyze from thousands of lines to a few hundred or less. It can refer the engineer to a specific API call, task, or log from a fixed time or window. This method's main challenge is distinguishing between normal variations in a log sequence and an anomalous variation.

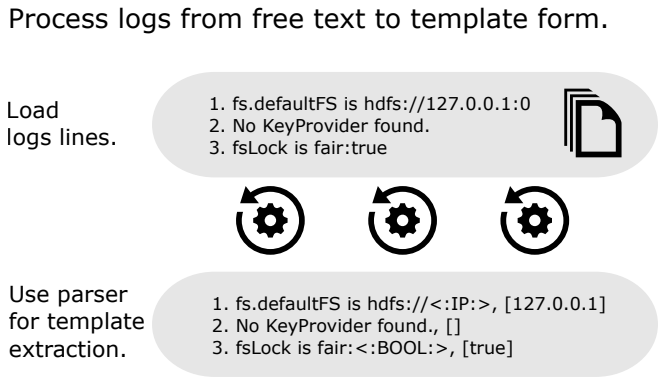

Figure 5. Process of parsing logs from free text to log template and variables.

This method is not easy, mainly because software is constantly being developed, new logs are added, and old ones are modified or deleted. Most of the methods learn from the past, i.e., from previous releases. In contrast, most failures occur during the early stages of new release development, where more data might be needed to learn release-specific normal and anomalous sequences. The work by Le et al. [4], in which logs were split chronologically, showed decreased performance in known methods. More work must be carried out to test the existing approach on logs from consecutive software releases.

4.1.1. Approaches

Statistical

Contemporary anomaly detection uses statistical, machine learning, or deep learning methods. Statistical methods can capture the relationship between log lines in one log file. They need much human-expert effort to select important features. Once selected, the elements are not portable to other systems when the process has to be repeated. Machine learning methods ease this process. A specific set of features, such as the sequence of log event IDs vectorized with the Term Frequency Inverse Document Frequency (TFIDF) method, is uniformly selected for all logs. TFIDF [32] is a widely known method that uses a fixed number of features to encode the importance of each word in a given text. Once the features have been collected, the model can be trained. However, when there is a change in the log event space, the length of the feature vector is also changed, and the model has to be retrained. This happens every time a new log event is added; the old one is deleted or modified. Deep learning models are more robust to changes in the log event space. Feature vectors are trained from log events using word2vec or Autoencoder.

One of the first anomaly detection methods was based on the statistical analysis of the features and rules. Xu et al. [33] proposed an unsupervised method that converts the log sequence into a count vector. Using PCA, anomalies are detected by checking whether the log sequence vector projected to a space spanning the first k PCA components is within normal bounds. Normal bounds are determined by analyzing normal historical executions. Invariant mining [34] uses program features observable in logs, which are always held during normal execution. For example, if the file is opened, it must be closed. It can uncover linear relationships between log events in an unsupervised manner. An event count matrix is built, where each row is the event count vector of a log line. Logsurfer [31] clusters log lines and applies predefined rules to them. Rule-based systems are not used today

due to their well-known limitations, but clustering is still being developed [35]. Another approach to leveraging statistical data is to build automata, and LogLens [36] keeps track of timestamps, log event parameters, content, and the number of occurrences. It then creates rules in an unsupervised manner, looking only at normal historical executions.

Machine Learning

Supervised machine learning (ML) methods including Support Vector Machines (SVM) [37], Decision Trees [38], Logistic Regression [39], and Random Forest [40,41] and unsupervised ML methods such as LogClustering [35], PCA [33], and invariant mining [34] take logs as input, extract their features, and train the classifier to label anomalous sequences. The logs are parsed to extract the log event ID in all methods. Source code is not used, but the line similarity is examined to determine log event templates. A fixed, sliding, or session window is applied to extract log sequences. The size of the window impacts the generalization capability of the model [41] and must be carefully selected. The final step of feature extraction is performed using a count vector. For the supervised ML methods, the model is trained on labeled vectors. The model will determine which one is anomalous when used on unseen logs.

Deep Learning

Statistical and machine learning approaches cannot deal with the change in log event space. Statistical methods use rules encoded by system experts or discovered from historical executions, and this does not enable them to recognize novel system behavior changes and distinguish normal from anomalous instances. Machine learning finds the relationships between data and results but uses a fixed representation of sequences. Every change in the log event space changes the dimensionality of the vector representation and forces the model to be retrained. On the other hand, deep learning methods learn the vector representation from raw data, which solves the problem of fixed feature representation in ML in many areas. However, in anomaly detection, some deep learning methods are also tightly coupled with feature representation [19,21,35,42,43]. Consequently, any change in the log event space reduces its precision and recall [8]. LSTM-based solutions [21,24,42] learn from the sliding window sequence of log events. If an unexpected log event ID is found, it is an anomaly. LogCNN [19] uses a matrix with a dimension equal to the log event space size. If the log event is absent during training, its value is undetermined. LogCluster [35] builds vectors with the length of the log event space size. The advantage of deep learning is the learned vector representation, which makes the model more robust to log event space changes, reducing the amount of model retraining. Log events are encoded not with the log event ID but with the vector that encodes the semantic meaning of the log event text. ADA [44] uses one-hot vectors for every word in the log template and learns to log event representations using LSTM. The distinction between normal and anomalous log lines is based on the threshold, an auto-adjusted parameter learned online during work. LogRobust [30] vectorizes log events with a weighted sum of word2vec vectors. It utilizes free text from the log event and vectorizes it with a pre-trained word2vec model. This approach was also adopted in [45]. Another application of word2vec can be found in LightLog [46], where the word2vec model is used to compress the sequence into a 300-dimension vector [47]. LogAnomaly [48] also combines words from log events with its algorithm, named template2vec, which is especially trained to encode specific synonyms and antonyms of log templates. LogBert [20] represents a log event as a sum of a randomly generated log key embedding and a position embedding generated with the sinusoid function.

4.1.2. Dealing with Concept Drift

The sophisticated, learned vector representations of words allow deep learning methods to deal with moderate log changes without model retraining. However, the log parsing tool still needs to be retrained, which is usually a time- and resource-consuming process.

Eventually, when the log event space is changed more significantly, the deep learning model has to be retrained. We have yet to find exact criteria for this, as none of the reviewed papers provide clear criteria. The bottleneck for most methods mentioned above is the log parsing algorithm. The number of open-source algorithms available online is increasing; among them, the most popular are Drain [49], Spell [50], LogParse [51], and nerlogparser [52]. To work well, they need fine- tuning and computational power. In addition, they cannot detect all the templates correctly, which propagates through the learning model and influences the results. To increase the accuracy, LogPunk [53] selects candidates based on punctuation marks; only then are templates extracted. There is also an effort to build a log parser taking semantic features into consideration [54] so that information about parameters is not omitted but is integrated into templates.

Some papers propose anomaly detection models without extracting log events. Brown et al. [55] do not clean the log lines at all. Logs are split into words or characters and fed into an RNN. Aussel et al. [56] use standard NLP techniques, including tokenization, stemming, and stop word removal, to extract log templates. Unigram, bigram, and trigram merge word2vec representations of words into one vector for a log line. Logsy [57] also uses text preprocessing known from NLP and removes digits and stop words. The log text is then tokenized and vectorized with the transformer encoder model. This also makes the model more robust to log event space changes. Farzad et al. [58], after similar preprocessing of the log text, proposed the usage of Autoencoder to vectorize the log text. Although promising, for both solutions, the anomaly detection results are worse than those of LogRobust and LogCNN, as presented in a survey [8].

4.1.3. Implementation

Most deep learning and machine learning methods are similar in implementation. Firstly, data need to be collected, and a large amount of normal and anomalous logs must be stored. Then, the data need to be labeled as normal and anomalous. The next step is extracting features from the logs so the algorithms can process them. The goal is to convert the log line text and variable lengths of log sequences into a fixed-length vector. The last step is to train the machine learning or deep learning model. The most expensive part in terms of implementation and computation is feature extraction. SVM [37], Decision Trees [38], Logistic Regression [39], and DeepLog [21] process log lines with the use of open-source tool Drain [49]. Then, log lines are converted into log event IDs, and a sliding window is used to deal with the variable length of the sequence. LogRobust [30] extracts log events using Drain, and then every line is vectorized separately, and every sequence is the weighted sum of the vector representation of the lines. Bertero et al. [59] uses standard NLP tools to remove all non-alphanumeric characters from logs. Every word in a line is vectorized with word2vec, but barycentring elements acquire lines and sequences. The line is the barycenter of every word, and the log sequence is the barycenter of all lines. LogLens [36] uses regular expressions to extract patterns. It does not require previous configurations or source code. LogLens uses finite-state automata to detect anomalies, and this also needs to be implemented and maintained as it is not available online. LogBert [20] is one of the few methods in which log parsing and model training and testing are available in the online implementation. It uses Drain to parse the logs and its own implementation of Bert to detect anomalies. The cost of implementation is low.

The implementation of [19,21,30,45,48] with log parsing is available online thanks to the authors of an empirical evaluation of the five most popular anomaly detection methods [4]. Thus, all these methods are easy to apply in a custom system. We wish to determine at which stage of software development this method is applicable. We consider two stages: integration testing and production. We will check if this is mentioned in the original paper or papers using the original method as a baseline. As in most cases, benchmark loghub is used [18]; we present reasoning about this benchmark, which allows us to infer the software development stage from the fact that this benchmark was used. The logs gathered there consist of logs from different applications deployed in

a production-like environment. This leads to the conclusion that the method tested in this benchmark suits production failures. As integration testing is aimed at simulating the production environment, it generates similar output to production logs. Historical data are also often stored and available. Thus, methods applicable to production logs should also work for integration logs. Our assumption is supported by work conducted by Yuan et al. [60], where Deeplog [21], designed as a production anomaly detection method, was used in integration testing logs, along with dedicated integration testing methods such as CAM [61]. The following methods [19,21,30,35,42,45,46,48,57,58] were tested on a benchmark [18] by the authors or in surveys [4,6,8]. As described above, the usage of benchmark loghub implies that the methods are suitable for logs from the production run and integration testing.

4.2. Retrieval-Based Methods

In many situations, more than information about an anomaly is needed. Anomaly detection can assign an anomaly label to a specific log file or log sequence but cannot reveal any meaningful information about it. While restricting several anomalous logs to analyze is beneficial for engineers, limiting the number of possible root causes would also be helpful. In some cases, it is sufficient for an engineer to assign a class of problems to the logs, and this information can be buried in logs but requires a different approach. Similarly, in a natural language, a different approach is needed in spam detection and sentiment analysis with more than two classes. In logs, failure categories are expected [62]. Such categories allow us to assign the problem directly to the correct maintenance group or code part. This approach is used during integration testing and performance issue resolution in leading software companies [61,63,64].

4.2.1. Approaches

CAM [61] and LogFaultFlagger [63] work on the test logs generated by the testing scripts. CAM was developed in Huaweii for integration test failures and assumes that the failure category can be determined by assigning the failure category of most similar failures from the past. The most similar historical failure is found by comparing the function points of the tests and measuring the cosine similarity between logs. TFIDF performs the vectorization of the test logs. CAM returns the failure category and the difference between the new and historical tests. If the cosine similarity is above the threshold, CAM uses 1-NN to determine the failure category; otherwise, KNN is used. The result indicates 58–65% correctly classified categories, depending on the dataset.

LogFaultFlagger was developed in Ericson as their attempt to use CAM failed. In its dataset, CAM achieved 50% accuracy. The developers use source code printing statements to process logs to remove the variable parts of log lines. They assume that the failure is visible in the difference between the normal and failed test log. Normal logs are found similarly to CAM by identifying logs with similar (cosine similarity) vector representations. This time, it is also TFIDF but is built on log event templates instead of English words. The same approach of using TFIDF at the logline level, along with the textual data, was used in Sarkar et al. [65] to assign the correct maintenance group at Ericsson. This has the obvious flaw of being an invalid representation after a change in log event space. Yuan et al. [60] and PerfSig [64] use standard NLP preprocessing techniques along with word2vec vectorization. Yuan et al. [60] label logs related to separate system tasks as failed or normal. Classifiers are ML methods implemented in the Scikit-learn library [66], and Random Forest achieves the best results. This method is distinguished from the previous solution because the classification results are the failed system tasks extracted from the system log with regular expressions. The method's accuracy is different from that of other methods known in the literature. Yan et al.'s [25] results are similar to those of the previous one study—namely, logs related to system tasks are labeled. The difference is in the way in which tasks are defined. Here, the system expert has to define and create system tests for them. The training dataset is created by running the set of created system

tests multiple times. Logs are preprocessed to extract log events and log event IDs. Then, similar log lines are clustered, where the timestamp is taken as a criterion. To vectorize the sequence of log events, the ParagraphVector is used. In the end, Multinomial Logistic Regression is used as a classifier.

4.2.2. Dealing with Concept Drift

CAM and LogFaultFlagger use TFIDF to vectorize log sequences. As mentioned before, this is not robust to new log templates as the length of the extracted vector will be changed, and the model would have to be retrained. Yuan et al. [25] use log events to build the matrix for log clustering. As the dimensions of a matrix are equal to the number of extracted events, it implies that the whole learning process has to be repeated after any change in log events. Yuan et al. [60] and PerfSig [64] overcome the limitation of the TFIDF vectorization of log lines and use standard NLP preprocessing techniques along with word2vec vectorization. The log line is represented as the average of words. PerfSig also clusters log lines that are semantically similar, which allows the grouping of logs from different threads. CAM and LogFaultFlagger cannot deal with concept drift, while Yuan et al.'s model [60] and PerfSig are more robust.

4.2.3. Implementation

CAM and LogFaultFlagger require a system expert to build the database with categorized historical failures. There is no online implementation as they are commercial tools. CAM achieved above 80% accuracy on their dataset, and LogFaultFlagger achieved 90% accuracy but was able to use its approach only on 62% of the bug reports. Yan et al.'s method [60], Yuan et al.'s method [25], and PerfSig are also based on the offline phase, where data related to categories are collected and labeled. Despite using standard NLP techniques and word2vec, all methods still need significant effort in implementing the main algorithm as there is no online publication. The main flaw of Yuan et al.'s method [25] is the effort needed from the system expert, who is required to build a knowledge base and implement an algorithm to extract log events and a machine learning model. The model achieved high precision and recall, above 97%. However, as in the previous method, there was no baseline method. Regarding computational complexity, the CAM study highlighted the inexpensive calculations, which could be counted in minutes. On the contrary, the CAM algorithm implemented in Ericson took 7 h to complete. Meanwhile, LogFaultFlagger needed 1 h to achieve the same task. Neither of these solutions is available online as they are commercial tools. Computational comparisons are inconclusive, as the datasets used differ from paper to paper. Yan et al.'s method [60] outpaces CAM and LogFaultFlagger in computation time by limiting the analysis to exception logs only. Their custom dataset's log processing took 37 min, while, for CAM, it required several hours. These methods can be used to detect integration test failures [61,63] and production and integration failures [25,60].

4.3. Execution Replay Methods

Anomaly detection and retrieval-based methods are high-level methods that return large parts of logs to the engineer labeled with anomaly or failure category class names. A lengthy process is needed to find the root cause, and knowledge of what was executed is crucial. The execution path is one of the most valuable pieces of information for the developer when dealing with a software failure. However, obtaining it for production or integration test failures is not easy. The usual method is to debug the process, but it is often impossible because the issue might not be reproducible or the client setup may not be accessible. This is the motivation for the exploration of other, noninvasive methods. As logs are often available and generated by the code, they present a part of the execution path. Several works seek to take advantage of this situation. This section will describe methods that retrieve the full or part of the execution path from the log.

The returned results have different levels of granularity, from execution statements [26,27] and files [67], through API calls [28], to the names of unit tests with a similar execution log [68]. Sometimes, it can be packaged with unit tests [28]. Reconstruction of the execution path plays a vital role during failure analysis. When done with high precision, it can help the developer to find the root cause of the failure. On the other hand, for long executions, the path will consist of thousands or millions of operations, which makes it impossible to use. This is a known flaw of SherLog [26].

4.3.1. Approaches

SherLog's goal is to infer what must, must not, and may have happened during the failed production run. It does so by combining information from the logs and source code. The result consists of an exact path and a probable path. LogMap [27] works similarly but uses only log fragments included in the bug report. This way, it overcomes the problem of too much information being returned. Similarly, Pathidea [67] reconstructs the execution path based on logs, but at the file level, using stack trace and textual information in a bug report. Pensieve [28] selects only calls from the API from the execution path. Then, it further reduces the number of API calls to a minimal subset of calls where the failure is reproduced and returns it packaged with a unit test. All these methods have the clear advantage of using source code and unit tests from failed releases. Our previous work [68] used a different approach to describe the execution path. The main idea is that logs from unit tests represent the underlying execution path. We can determine the execution path from the runtime by matching the unit test suite logs to the runtime execution log segmented with a sliding window or fixed window. In this way, the runtime receives meaningful labels and unit test suite names, which can be treated as an approximation of the execution path for the test suite coverage. On the other hand, logs from unit tests can be treated as labeled sets of logs, which, in previous works, was done manually [69]. They are grouped into categories by the originating test suite. Log preprocessing is done by Drain, and the log event ID sequence is then vectorized by TFIDF and classified with Random Forest. It is worth mentioning that this approximation is not as accurate as that of SherLog, but it is much faster, because the most demanding part in terms of computation is the training, which is done once; then, for every runtime, the log computation time is equal to the processing and vectorization time of the runtime log and Random Forest classification time, which is fast. Labeling a group of logs also means reducing the amount of information to analyze by the programmer, and unit test suite names are more meaningful to the developer than raw logs.

4.3.2. Dealing with Concept Drift

All mentioned methods depend only on the source code relevant to the release where the failure is visible, and it is done directly or indirectly by using logs from unit tests. This approach is robust to concept drift as it extracts all relevant information from the source code, not from historical data.

4.3.3. Implementation

Sherlog, Pensieve, LogMap, and Pathidea do not have an online implementation; they are non-trivial algorithms. Dobrowolski et al.'s implementation is available online on GitHub. Sherlog has known the issue of state explosion by analyzing every possible path of the execution. Other tools avoid this problem by reducing the search space to logs included in the bug report [27] or changing the granularity of the result to a higher level, such as API calls [28] or unit test suite names [68]. All methods can be used to deal with production and integration test failures. Regarding the computation time, SherLog's flaw is that it suffers from state explosion, as it examines every possible execution path, which is infeasible for an extensive software system. LogMap and Pensieve avoid this by reducing the search space. Dobrowolski's method relies on approximations made by the ML model

so that the computation cost is almost entirely moved to the training phase. However, it still requires parser computations to extract the log event ID sequence.

5. Discussion—A Cheat Sheet for Engineers

In this section, we gather all the information in the form of a Table 2 so that the the engineer can easily identify an interesting algorithm and check and compare its features.

Table 2. Automated log analysis methods. The online column describes whether there is an online implementation; algorithm (statistic, automated, machine learning, or deep learning) describes which algorithm is used for classification. The applicable stage indicates at which stage of software development the method can be used. The result is the type of returned value, and F1 is the score of the method presented on a public dataset (if available) or a custom dataset.

Method	Online	Model	Applicable Stage	Concept Drift	Result	F1 Score
IM [34]	✓	statistic features	int&prod	retrain	anomaly	91%
PCA [33]	✓	statistic features	int&prod	retrain	anomaly	55%
LogSurfer [31]		statistic features	int&prod	retrain	anomaly	N/A
SVM [37]	✓	ML	int&prod	retrain	anomaly	85%
LogEvent2vec [40]		ML	int&prod	retrain	anomaly	83%
Logistic Regression [39]	✓	ML	int&prod	retrain	anomaly	82%
DT [38]	✓	ML	int&prod	retrain	anomaly	74%
LogCluster [35]	✓	ML	production	retrain	anomaly	57%
LogLens [36]		Automata	int&prod	relearn	anomaly	97%
LogRobust [30]	✓	DL	int&prod	robust	anomaly	99.5%
LightLog [46]	✓	DL	int&prod	robust	anomaly	97%
LogBert [20]	✓	DL	int&prod	robust	anomaly	96.64%
CNN [19]	✓	DL	int&prod	relearn	anomaly	90.8%
Logsy [57]	✓	DL	int&prod	robust	anomaly	80%
Farzad et al. [58]	✓	DL	int&prod	robust	anomaly	80%
ClCLSTM [42]		DL	int&prod	relearn	anomaly	76%
PLELog [45]	✓	DL	int&prod	relearn	categories	74.4%
LogAnomaly [48]	✓	DL	int&prod	robust	anomaly	48.3%
DeepLog [21]	✓	DL	int&prod	relearn	anomaly	42.6%
Yuan et al. [25]		ML	int&prod	relearn	task name	97%
LFF [63]		ML	integration	relearn	category	88%
CAM [61]		ML	integration	robust	category	58%
SherLog [26]		static	int&prod	relearn	execution path	N/A
Pensieve [28]		static	int&prod	relearn	API calls	N/A
Dobrowolski et al. [68]	✓	ML	int&prod	relearn	labeled execution path	N/A

We utilize categories important to the engineer: online availability, the model used, the applicable stage of software development, robustness to log changes, and F1 score. The methods are arranged in groups: anomaly detection, retrieval-based, and execution replay methods. A bold line separates the groups. In every group, the methods are sorted by the F1 score. The availability of online tools is one of the most important factors, as applying the model to satsify customer needs is crucial. The model category is used to distinguish between statistical, ML, and DL approaches. The appropriate stage category describes at

which stage of software development the tool can be used. The methods included in this survey can be applied to integration testing and production failures.

Concept drift is a well-known definition of the problem of log changes due to a normal development process [8]. For commercial implementation, it is crucial to retrain the whole process every time the log event space is changed or only once in a while; thus, a separate category in the next column of the table takes two values: 'retrain' or 'robust'. The 'retrain' value means that the method has to be trained from the beginning every time there is a log change. On the other hand, 'robust' implies that it can operate to some degree with such changes. It is worth mentioning that none of the present methods is entirely immune to concept drift, and it always reduces the model's performance and retraining is needed eventually.

Finally, the result column shows what is returned to the end user, and the F1 score column shows the performance result in this task. When possible, we use scores gathered in review publications with their implementations [4,7,8], as they can be significantly different from the ones presented in the original paper. For example, DeepLog is said to have an F1 score of 96% in the original article [21], while, in the implementation of LogADEmpirical [4], it is only 42%. The reason for this is that training test selection was performing randomly in the original paper, while, in real-life scenarios, we have to consider chronological features. The same situation applies with LogAnomaly.

We should also note the approach presented in DeepLog and LogAnomaly, because they detect features much faster than LogRobust and LogCNN [19]. It should be mentioned that some methods within a group have the same features and similar F1 scores. This area uses machine learning methods: SVM, LogEvent2Vec, and Logistic Regression (LR). SVM is superior to LR when learning non-separable features linearly. On the other hand, Logevent2Vec uses the computationally intensive word2vec to extract features, while still being worse than SVM. While retraining is inevitable for log changes, it is a reason to use lighter computations. Thus, LogRobust, LightLog, and LogBert are very similar methods and it is difficult to determine which one is superior. Although LogRobust was often implemented in review publications [4,8], the results of the original article were confirmed. This is the reason that it is safer to rely on LogRobust's results.

The engineer has to consider three parts of the implementation: log parsing, log sequence vectorization, and the model. Log parsing is mostly a choice between the Drain and NLP preprocessing techniques. Drain processing introduces another level of uncertainty by losing a small percentage of the information about log lines. Drain has to be trained before usage and requires retraining every time there is a change in log events. It is important to remember that training is very time-consuming; it can take hours when the size of the log corpus is large, such as 20 GB of Windows logs; the current implementation needs to be parallelized.

However, NLP techniques do not require any training. Meanwhile, they do not provide information on log event templates. We believe thatt NLP techniques should be preferred before log template extraction, as they are more generic and less time-consuming. Log vectorization is semantic, as is count vectorization. Semantic vectorization keeps the information about the sequence and is more robust to concept drift, so it should be preferred whenever possible. Count vectorization is faster and can be chosen when log events are not often changed, so there is a minimal risk of full model retraining. Count vectorization can only be used when combined with extracted log event templates. Finally, the choice of the model should be based on the available F1 score. In the reviews, it is clear that recurrent neural networks with semantic vectorization achieve the best results. For quantity vectorization, Random Forest is the first choice. It is important to remember that there is no technique to prevent retraining so far. There are only techniques that require less frequent retraining.

To summarize the difficulty of implementation, we propose a three-degree scale: easy, medium, and hard. Easy implementation indicates a method with an existing complete and working online implementation; medium reflects that the method does have partial

online implementation or pseudo-code presented in the paper, and hard indicates that there is only a description of the algorithm, without online implementation or pseudo-code. Results are gathered in Table 3. For example, LogSurfer [31], as a system without online implementation, requires great effort to implement, and a rule-based system requires great effort to configure and maintain. IM [34] also does not have online implementation; thus, the cost is high, although, by automatically retrieving features from logs, the cost of configuration is lowered. This method retrieves invariants from relations between log events; while there is no consideration of the concept of drift, this will increase the cost of maintenance. In the case of PCA [33], there is a lack of online implementation. The next developer must choose the features that he wishes to extract from the logs, which increases the cost of configuration. The cost of maintenance is increased because such features might have changed during the software's lifetime. With regard to machine learning methods, SVM [37], LR [39], and DT [38] require a carefully selected set of features. Implementation of ML algorithms can be taken from well-known libraries such as [66]. Thus, their implementation is low, the configuration is high, and maintenance is high due to a lack of concept drift consideration. In the case of deep learning methods, they are learned to extract features for logs. However, the logs have to be preprocessed nonetheless. This keeps the cost of configuration high as log preprocessing is based on a regular expression. The implementation cost is set to medium, as they are rarely raw algorithms such as Bi-LSTM, available in online libraries, but are more complicated and sophisticated. The maintenance cost is low only in a few examples, such as [30,68], as they are robust to concept drift.

Often, online implementations still need to be completed. For example, Ref. [4] lacks the log parsing part; LogRobust [30] contains only the Jupyter notebook with the model, without log parsing and log vectorization. The existing online solutions presented in Table 4 are proofs of concept and are not optimized for extensive data, so engineers should consider building their implementation based on existing hints in online implementations.

Table 3. Cost of method implementation divided into three categories: cost of code implementation, cost of the configuration, cost of maintenance. The cost of code implementation is low when there is online implementation. On the other hand, there are methods with no online implementation or even pseudo-code presented. The cost of the configuration is high when, for example, the expert must create a set of regular expressions for Drain to work on their logs. The cost of configuration is low when there is very little required besides running the code. The cost of maintenance is high when the method is based on regular expressions, which need to be updated every time the logs are changed. The overall cost is the average of the costs from the previous columns.

Method	Impl. Cost	Conf. Cost	Mainten. Cost	Overall Cost
IM [34]	high	low	high	**medium**
PCA [33]	high	medium	high	**high**
LogSurfer [31]	high	high	high	**high**
SVM [37]	low	high	high	**high**
LogEvent2vec [40]	high	high	high	**high**
Logistic Regression [39]	low	high	high	**high**
DT [38]	low	high	high	**high**
LogCluster [35]	low	high	high	**high**
LogLens [36]	high	high	high	**high**
LogRobust [30]	medium	high	high	**high**
LightLog [46]	high	high	high	**high**
LogBert [20]	medium	high	high	**high**
CNN [19]	low	high	high	**high**
Logsy [57]	high	high	high	**high**

Table 3. *Cont.*

Method	Impl. Cost	Conf. Cost	Mainten. Cost	Overall Cost
Farzad et al. [58]	low	high	high	**high**
ClCLSTM [42]	medium	high	high	**high**
PLELog [45]	low	high	high	**high**
LogAnomaly [48]	low	high	high	**high**
DeepLog [21]	low	low	high	**high**
Yuan et al. [25]	high	high	high	**high**
LFF [63]	high	high	high	**high**
CAM [61]	high	high	high	**high**
SherLog [26]	high	high	high	**high**
Pensieve [28]	high	high	high	**high**
Dobrowolski et al. [68]	low	high	low	**medium**

Table 4. Online toolkits with method implementation.

Source Code	Method(s)	Access Date
https://github.com/logpai/loglizer	SVM [37], DT [38], LogisticRegression [39], PCA [33], IM [34], LogCluster [35]	15 May 2023
https://github.com/logpai/deep-loglizer	DeepLog [21], LogAnomaly [48], Logsy [57], Autoencoder [58], LogRobust [30], LogCNN [19]	15 May 2023
https://github.com/LogIntelligence/LogADEmpirical	PLELog [45], DeepLog, LogAnomaly, LogCNN, LogRobust	15 May 2023
https://github.com/Aquariuaa/LightLog	LightLog [46]	15 May 2023
https://github.com/HelenGuohx/logbert/	LogBert [20]	15 May 2023
https://github.com/dobrowol/defects4all	Dobrowolski et al. [68]	15 May 2023

6. Conclusions

Automated log analysis is a growing field of research and is attracting more attention from the industry. Some practical features have to be taken into account.

Firstly, and most importantly, the tool has to be able to work with up-to-date releases and return results for logs that are changed due to the normal development process. Training, validation, and testing datasets should be split chronologically to ensure this. The conclusion is that an online dataset should be built to reflect this need. Not all available log data in the popular loghub [18] have a timestamp, and those that have timestamps do not have explicit information about software updates during log collection.

Many methods treat log sequences as a type of language, and, on this basis, they apply NLP methods. However, it is still unclear whether this abstraction is valid, as the features of this language were not studied in any research paper. One feature of a log sequence that is particularly uncommon for any language is the speed of changes in vocabulary. Single log lines and all sequences might change from release to release, i.e., at least a few times a year, which is unprecedented in natural languages. The method should detect which changes in logs are due to normal development processes and which are anomalous. The rate of change also puts historical logs into question. It would be very informative to determine whether logs from previous years are valuable in determining anomalies in the newest release.

The methods presented were also not tested against the different densities of log statements in the source code. However, this is a crucial feature of the software when dealing with anomalous behavior. Software with one log printing statement every 50 lines differs from software with one in a thousand lines for automated log analysis.

The methods should be less dependent on preprocessing steps. Parsing logs to log events to extract the execution path is costly in terms of time and memory complexity. They are often system-specific. Deep learning can extract features from raw data. It should be used when minimizing the impact of preprocessing logs. It can also reduce the dependency of the method on expert knowledge.

To summarize, we propose the following directions for further research.

The first direction is to build a universal benchmark of logs from consecutive releases that should be created for concept drift research; the second direction is to examine whether it is possible to build an ideal, robust model requiring no retraining; this relates to the third direction, the question of the relation between the age of the logs and future failures; the fourth direction is the relation between the density of log printing statements in the code and the result of the method; the last direction is minimizing the impact of log template extraction on the inaccuracy of the method.

Author Contributions: Conceptualization, W.D., M.N. and O.U.; Supervision, M.N. and O.U.; Methodology, M.N. and O.U.; Writing—original draft, W.D.; Writing—review and editing, W.D., M.N. and O.U. All authors have read and agreed to the published version of the manuscript.

Funding: The Polish Ministry of Education and Science financed this work. Funds were allocated from the "Implementation Doctorate" program.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ML	Machine Learning
DL	Deep Learning

References

1. Yuan, W.; Lu, S.; Sun, H.; Liu, X. How are distributed bugs diagnosed and fixed through system logs? *Inf. Softw. Technol.* **2020**, *119*, 106234. [CrossRef]
2. He, S.; He, P.; Chen, Z.; Yang, T.; Su, Y.; Lyu, M.R. A survey on automated log analysis for reliability engineering. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–37. [CrossRef]
3. Yang, N.; Cuijpers, P.; Schiffelers, R.; Lukkien, J.; Serebrenik, A. An Interview Study of how Developers use Execution Logs in Embedded Software Engineering. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Madrid, Spain, 25–28 May 2021; pp. 61–70. [CrossRef]
4. Le, V.H.; Zhang, H. Log-based anomaly detection with deep learning: How far are we? In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 22–27 May 2022; pp. 1356–1367.
5. Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, Bari, Italy, 26–27 June 2008; pp. 68–77.
6. Zhao, X.; Jiang, Z.; Ma, J. A Survey of Deep Anomaly Detection for System Logs. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022; pp. 1–8. [CrossRef]
7. He, S.; Zhu, J.; He, P.; Lyu, M.R. Experience report: System log analysis for anomaly detection. In Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada, 23–27 October 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 207–218.
8. Chen, Z.; Liu, J.; Gu, W.; Su, Y.; Lyu, M.R. Experience report: Deep learning-based system log analysis for anomaly detection. *arXiv* **2021**, arXiv:2107.05908.
9. Avizienis, A.; Laprie, J.C.; Randell, B.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **2004**, *1*, 11–33. [CrossRef]
10. Brar, H.K.; Kaur, P.J. Differentiating integration testing and unit testing. In Proceedings of the 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 11–13 March 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 796–798.
11. Liu, H.; Lu, S.; Musuvathi, M.; Nath, S. What bugs cause production cloud incidents? In Proceedings of the Workshop on Hot Topics in Operating Systems, Bertinoro, Italy, 13–15 May 2019; pp. 155–162.

12. Murali, V.; Gross, L.; Qian, R.; Chandra, S. Industry-scale IR-based bug localization: A perspective from Facebook. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Madrid, Spain, 25–28 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 188–197.
13. Wong, W.E.; Debroy, V.; Li, Y.; Gao, R. Software fault localization using dstar (d*). In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, Gaithersburg, MD, USA, 20–22 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 21–30.
14. Wen, M.; Chen, J.; Tian, Y.; Wu, R.; Hao, D.; Han, S.; Cheung, S.C. Historical spectrum based fault localization. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2348–2368. [CrossRef]
15. Leucker, M.; Schallhart, C. A brief account of runtime verification. *J. Log. Algebr. Program.* **2009**, *78*, 293–303. [CrossRef]
16. Krichen, M. How Artificial Intelligence Can Revolutionize Software Testing Techniques. In *Proceedings of the Innovations in Bio-Inspired Computing and Applications: Proceedings of the 13th International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA 2022) Held During 15–17 December 2022*; Springer: Cham, Switzerland, 2023; pp. 189–198.
17. Lima, R.; da Cruz, A.M.R.; Ribeiro, J. Artificial intelligence applied to software testing: A literature review. In Proceedings of the 2020 15th Iberian Conference on Information Systems and Technologies (CISTI), Sevilla, Spain, 24–27 June 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6.
18. He, S.; Zhu, J.; He, P.; Lyu, M.R. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv* **2020**, arXiv:2008.06448.
19. Lu, S.; Wei, X.; Li, Y.; Wang, L. Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network. In Proceedings of the 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), Athens, Greece, 12–15 August 2018; pp. 151–158. [CrossRef]
20. Guo, H.; Yuan, S.; Wu, X. Logbert: Log anomaly detection via bert. In Proceedings of the 2021 International Joint Conference on Neural Networks (IJCNN), Shenzhen, China, 18–22 July 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–8.
21. Du, M.; Li, F.; Zheng, G.; Srikumar, V. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In Proceedings of the CCS'17 2017 ACM SIGSAC Conference on Computer and Communications Security; Association for Computing Machinery, Dallas, TX, USA, 30 October–3 November 2017; pp. 1285–1298. [CrossRef]
22. Lee, Y.; Kim, J.; Kang, P. LAnoBERT: System log anomaly detection based on BERT masked language model. *arXiv* **2021**, arXiv:2111.09564.
23. van der Aa, H.; Rebmann, A.; Leopold, H. Natural language-based detection of semantic execution anomalies in event logs. *Inf. Syst.* **2021**, *102*, 101824. [CrossRef]
24. Chen, Y.; Luktarhan, N.; Lv, D. LogLS: Research on System Log Anomaly Detection Method Based on Dual LSTM. *Symmetry* **2022**, *14*, 454. [CrossRef]
25. Yuan, Y.; Anu, H.; Shi, W.; Liang, B.; Qin, B. Learning-based anomaly cause tracing with synthetic analysis of logs from multiple cloud service components. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; IEEE: Piscataway, NJ, USA, 2019; Volume 1, pp. 66–71.
26. Yuan, D.; Mai, H.; Xiong, W.; Tan, L.; Zhou, Y.; Pasupathy, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, PA, USA, 13–17 March 2010; pp. 143–154.
27. Chen, A.R. An empirical study on leveraging logs for debugging production failures. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 25–31 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 126–128.
28. Zhang, Y.; Makarov, S.; Ren, X.; Lion, D.; Yuan, D. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; pp. 19–33.
29. Amusuo, P.C.; Sharma, A.; Rao, S.R.; Vincent, A.; Davis, J.C. Reflections on Software Failure Analysis. *arXiv* **2022**, arXiv:2209.02930.
30. Zhang, X.; Xu, Y.; Lin, Q.; Qiao, B.; Zhang, H.; Dang, Y.; Xie, C.; Yang, X.; Cheng, Q.; Li, Z.; et al. Robust log-based anomaly detection on unstable log data. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 807–817.
31. Prewett, J.E. Analyzing Cluster Log Files Using Logsurfer. Citeseer. 2003. Available online: https://www.semanticscholar.org/paper/Analyzing-cluster-log-files-using-Logsurfer-Prewett/d9a2a773348e6dc1c0bef303cf188145267bd8c1 (accessed on 15 May 2023).
32. Salton, G.; Buckley, C. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.* **1988**, *24*, 513–523. [CrossRef]
33. Xu, W.; Huang, L.; Fox, A.; Patterson, D.; Jordan, M.I. Detecting large-scale system problems by mining console logs. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 117–132.
34. Lou, J.G.; Fu, Q.; Yang, S.; Xu, Y.; Li, J. Mining invariants from console logs for system problem detection. In Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10), Boston, MA, USA, 23–25 June 2010.

35. Lin, Q.; Zhang, H.; Lou, J.G.; Zhang, Y.; Chen, X. Log clustering based problem identification for online service systems. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Austin, TX, USA, 14–22 May 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 102–111.
36. Debnath, B.; Solaimani, M.; Gulzar, M.A.G.; Arora, N.; Lumezanu, C.; Xu, J.; Zong, B.; Zhang, H.; Jiang, G.; Khan, L. LogLens: A real-time log analysis system. In Proceedings of the 2018 IEEE 38th international conference on distributed computing systems (ICDCS), Vienna, Austria, 2–6 July 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1052–1062.
37. Liang, Y.; Zhang, Y.; Xiong, H.; Sahoo, R. Failure prediction in ibm bluegene/l event logs. In Proceedings of the Seventh IEEE International Conference on Data Mining (ICDM 2007), Omaha, Nebraska, 28–31 October 2007; IEEE: Piscataway, NJ, USA, 2007; pp. 583–588.
38. Chen, M.; Zheng, A.X.; Lloyd, J.; Jordan, M.I.; Brewer, E. Failure diagnosis using decision trees. In Proceedings of the International Conference on Autonomic Computing, 2004. Proceedings, New York, NY, USA, 17–19 May 2004; IEEE: Piscataway, NJ, USA, 2004; pp. 36–43.
39. Bodik, P.; Goldszmidt, M.; Fox, A.; Woodard, D.B.; Andersen, H. Fingerprinting the datacenter: Automated classification of performance crises. In Proceedings of the 5th European Conference on Computer Systems, Paris, France, 13–16 April 2010; pp. 111–124.
40. Wang, J.; Tang, Y.; He, S.; Zhao, C.; Sharma, P.K.; Alfarraj, O.; Tolba, A. LogEvent2vec: LogEvent-to-vector based anomaly detection for large-scale logs in internet of things. *Sensors* **2020**, *20*, 2451. [CrossRef] [PubMed]
41. Ryciak, P.; Wasielewska, K.; Janicki, A. Anomaly Detection in Log Files Using Selected Natural Language Processing Methods. *Appl. Sci.* **2022**, *12*, 5089. [CrossRef]
42. Yen, S.; Moh, M.; Moh, T.S. Causalconvlstm: Semi-supervised log anomaly detection through sequence modeling. In Proceedings of the 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), Boca Raton, FL, USA, 16–19 December 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1334–1341.
43. Studiawan, H.; Sohel, F.; Payne, C. Anomaly detection in operating system logs with deep learning-based sentiment analysis. *IEEE Trans. Dependable Secur. Comput.* **2020**, *18*, 2136–2148. [CrossRef]
44. Yuan, Y.; Adhatarao, S.S.; Lin, M.; Yuan, Y.; Liu, Z.; Fu, X. Ada: Adaptive deep log anomaly detector. In Proceedings of the IEEE INFOCOM 2020-IEEE Conference on Computer Communications, Toronto, ON, Canada, 6–9 July 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 2449–2458.
45. Yang, L.; Chen, J.; Wang, Z.; Wang, W.; Jiang, J.; Dong, X.; Zhang, W. Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 25–28 May 2021; pp. 1448–1460. [CrossRef]
46. Wang, Z.; Tian, J.; Fang, H.; Chen, L.; Qin, J. LightLog: A lightweight temporal convolutional network for log anomaly detection on the edge. *Comput. Netw.* **2021**, *203*, 108616. [CrossRef]
47. Joulin, A.; Grave, E.; Bojanowski, P.; Douze, M.; Jégou, H.; Mikolov, T. Fasttext. zip: Compressing text classification models. *arXiv* **2016**, arXiv:1612.03651.
48. Meng, W.; Liu, Y.; Zhu, Y.; Zhang, S.; Pei, D.; Liu, Y.; Chen, Y.; Zhang, R.; Tao, S.; Sun, P.; et al. LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In Proceedings of the IJCAI, Macao, China, 10–16 August 2019; Volume 19, pp. 4739–4745.
49. He, P.; Zhu, J.; Zheng, Z.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. In Proceedings of the 2017 IEEE international conference on web services (ICWS), Honolulu, HI, USA, 25–30 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 33–40.
50. Du, M.; Li, F. Spell: Streaming parsing of system event logs. In Proceedings of the 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 12–15 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 859–864.
51. Meng, W.; Liu, Y.; Zaiter, F.; Zhang, S.; Chen, Y.; Zhang, Y.; Zhu, Y.; Wang, E.; Zhang, R.; Tao, S.; et al. Logparse: Making log parsing adaptive through word classification. In Proceedings of the 2020 29th International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 3–6 August 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–9.
52. Studiawan, H.; Sohel, F.; Payne, C. Automatic log parser to support forensic analysis. In Proceedings of the 16th Australian Digital Forensics Conference, Joondalup, Australia, 4–5 December 2018.
53. Zhang, S.; Wu, G. Efficient Online Log Parsing with Log Punctuations Signature. *Appl. Sci.* **2021**, *11*, 11974. [CrossRef]
54. Huo, Y.; Su, Y.; Li, B.; Lyu, M.R. SemParser: A Semantic Parser for Log Analysis. *arXiv* **2021**, arXiv:2112.12636.
55. Brown, A.; Tuor, A.; Hutchinson, B.; Nichols, N. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In Proceedings of the First Workshop on Machine Learning for Computing Systems, Tempe, AZ, USA, 12 June 2018; pp. 1–8.
56. Aussel, N.; Petetin, Y.; Chabridon, S. Improving performances of log mining for anomaly prediction through nlp-based log parsing. In Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, 25–28 September 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 237–243.
57. Nedelkoski, S.; Bogatinovski, J.; Acker, A.; Cardoso, J.; Kao, O. Self-attentive classification-based anomaly detection in unstructured logs. In Proceedings of the 2020 IEEE International Conference on Data Mining (ICDM), Sorrento, Italy, 17–20 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1196–1201.

58. Farzad, A.; Gulliver, T.A. Unsupervised log message anomaly detection. *ICT Express* **2020**, *6*, 229–237. [CrossRef]
59. Bertero, C.; Roy, M.; Sauvanaud, C.; Tredan, G. Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection. In Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 23–26 October 2017; pp. 351–360. [CrossRef]
60. Yuan, Y.; Shi, W.; Liang, B.; Qin, B. An approach to cloud execution failure diagnosis based on exception logs in openstack. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), San Diego, CA, USA, 25–30 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 124–131.
61. Jiang, H.; Li, X.; Yang, Z.; Xuan, J. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 712–723.
62. Mi, H.; Wang, H.; Zhou, Y.; Lyu, M.R.T.; Cai, H. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 1245–1255. [CrossRef]
63. Amar, A.; Rigby, P.C. Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 140–151. [CrossRef]
64. He, J.; Lin, Y.; Gu, X.; Yeh, C.C.M.; Zhuang, Z. PerfSig: Extracting performance bug signatures via multi-modality causal analysis. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 22–27 May 2022; pp. 1669–1680.
65. Sarkar, A.; Rigby, P.C.; Bartalos, B. Improving Bug Triaging with High Confidence Predictions at Ericsson. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 29 September–4 October 2019; pp. 81–91. [CrossRef]
66. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
67. Chen, A.R.; Chen, T.H.; Wang, S. Pathidea: Improving Information Retrieval-Based Bug Localization by Re-Constructing Execution Paths Using Logs. *IEEE Trans. Softw. Eng.* **2022**, *48*, 2905–2919. [CrossRef]
68. Dobrowolski, W.; Nikodem, M.; Zawistowski, M.; Unold, O. Improved Software Reliability Through Failure Diagnosis Based on Clues from Test and Production Logs. In Proceedings of the International Conference on Dependability and Complex Systems, Hiroshima, Japan, 26–30 March 2022; Springer: Cham, Switzerland, 2022; pp. 42–49.
69. Bose, R.J.C.; van der Aalst, W.M. Discovering signature patterns from event logs. In Proceedings of the 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), Singapore, 16–19 April 2013; IEEE: Piscataway, NJ, USA 2013; pp. 111–118.