Log Based Fault Localization with Unsupervised Log Segmentation

Wojciech Dobrowolski * , Kamil Iwach-Kowalski , Maciej Nikodem , Olgierd Unold

Posted Date: 15 August 2024

doi: 10.20944/preprints202408.1096.v1

Keywords: Automated Log Analysis; Log-based Fault Localization; Log Sequence; Unsupervised Log Sequence Segmentation; Software Reliability

 doi:10.20944/preprints202408.1096.v1

Article

Log Based Fault Localization with Unsupervised Log Segmentation

Wojciech Dobrowolski [1,2,*], Kamil Iwach-Kowalski [1], Maciej Nikodem [2] and Olgierd Unold [2]

[1] Nokia, Rodziny Hiszpanskich 8, 02-685 Warszawa, Poland; kamil.1.kowalski@nokia.com (K.I.K.)
[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: Localizing faults in a software is a tedious process. The manual approach is becoming impractical because of the large size and complexity of contemporary computer systems as well as their logs, which are often the primary source of information about the fault. Log-based Fault Localization (LBFL) is a popular method applied for this purpose. However, in real-world scenarios, this method is vulnerable to a large number of previously unseen log lines. In this paper, we propose a novel method that can guide programmers to the location of a fault by creating a hierarchy of log lines with the highest rank, selected by the traditional LBFL method. We use the intuition that the symptoms of faults are in the context of normal behavior, whereas suspicious log lines grouped together are from new or additional functionalities turned on during faulty execution. To obtain this context, we used unsupervised log sequence segmentation, which has been previously used to segment log sequences into meaningful segments. Experiments on real-life examples show that our method reduces the effort to find the most crucial logs by up to 64% compared to the traditional timestamp approach. We demonstrated that context is highly useful in advancing fault localization, showing the possibility of further speeding up the process.

Keywords: automated log analysis; log-based fault localization; log sequence; unsupervised log sequence segmentation; software reliability

1. Introduction

Software systems play a crucial role in modern societies. They are responsible for services delivered by airports, banks, medical centers, and even the government. The need for reliability has grown rapidly. In July 2024 a software glitch in the CrowdStrike patch, a leading cybersecurity company, caused a major outage for Microsoft system users, leaving many organizations without a working operating system. Flights in the main airports were grounded, payments were blocked, and the functioning of medical centers was paralyzed. This incident highlighted the profound impact of software failures on businesses and underscored the importance of effective fault localization in software systems. Fault localization is a critical aspect of software maintenance and reliability. It plays a pivotal role in reducing the time and resources required to debug and fix issues, thereby ensuring the smooth functioning of complex systems.

Fault localization works by narrowing the scope of a developer's interest from the entire codebase to the specific areas where the fault is likely to be located. This process typically involves two main stages: detection of anomalies and correlation with the potential place of the fault or root cause. Anomalies are identified using various data sources, such as logs, execution traces, and metrics. Once detected, these anomalies were analyzed to determine the patterns and correlations that point to the probable fault location. By effectively pinpointing these locations, fault localization techniques can help developers focus their debugging efforts more precisely, thereby accelerating the process of resolving software issues. Fault localization often relies on multimodal data, including logs, execution traces, metrics [1], test results, and textual information [2]. To obtain execution traces and metrics, the system must be instrumented in advance, that is, it must be equipped with a code that generates the mentioned data. A popular framework for this purpose is OpenTelemetry [3], which implements the

 doi:10.20944/preprints202408.1096.v1

concept of observability and involves understanding the internal state of a system through its outputs. Such an enriched system is a good foundation for applying automatic diagnosis and localization methods, such as AIOps [4]. However, obtaining such a rich source of data from a system is infrequent and sometimes not feasible. Thus, many fault-localization methods rely on unimodal data, that is, data from only one source, such as logs.

Spectrum-Based Fault Localization (SBFL) [5] is a rapidly growing field that use unimodal data. The methods based on code coverage (spectrum) are lightweight and competitive. They are fast and scalable. They use code coverage generated by module and unit tests, which are often collected during execution. The anomaly measure was calculated based on the number of times a line appeared in the passed and failed tests. The correlation is such that the lines of code that appear more frequently in failed tests are more likely to be the source of the problem. Statistical methods [6–8] are frequently used to calculate the suspicious scores. Based on this metric, a ranking of program lines can be produced and suggested to the developer regarding where to look first. Ideally, the line with the actual error should be at the top of the suggestions.

Machine learning [9] and deep learning [10,11] models also use code coverage to predict the faulty lines. Training involves teaching the network to predict the test results based on code coverage. Then, during inference, virtual tests consisting of a single line of code are inputted. The returned results indicate that the line is a potential source of error. Deep methods use code coverage, represented as a Coverage Matrix [10,12]. It is important that the code coverage of similar tests is placed next to each other in the matrix, as the CNNs used in these solutions excel at detecting local dependencies. This allows the network to learn which changes in test execution cause an error.

However, code coverage information alone does not always correlate with the actual cause of an error. This is because the execution frequency data of a given line affects the localization result. Non-faulty code fragments can be executed more frequently than actual faulty fragments, which skews the SBFL result. Therefore, more costly and powerful Slicing-Based Fault Localization methods are used to solve this problem.

Slicing methods identify exact instructions for a given system output. The reference point is a specific instruction or variable that we want to trace. The flow is observed throughout all places, from its creation and modification to its return. Slicing can be performed statically [13,14] or dynamically [15–17], or by combining both approaches [18] to limit the size of static slicing while improving the quality of dynamic slicing.

Obtaining code coverage or instrumenting code to obtain traces is not always feasible because code coverage is not collected during field execution, and instrumentation of the code often produces an unacceptable level of overhead and may introduce faults themselves. Logs are a source of information that introduces minimal overhead and intervention in the system. Therefore, logs are the most common source of information after failure. As part of the execution path, logs can be used as a source of fault localization information. Logs are generated by lines in the source code and are placed by programmers to reflect a specific intention. The structure and consistency of logs can vary significantly depending on the discipline within a company [19]. This variation necessitates a high resilience in log-based methods. Logs can be considered in terms of their static and dynamic components, where the static part corresponds to the text placed in the code by the programmer, whereas the dynamic part corresponds to the text generated during the program's execution.

Several fault localization methods based on logs have been proposed. Some of these methods use logs in conjunction with other data such as the content of a build configuration file [20] or key performance indicators (KPIs) [21]. However, it is also possible to use logs alone. Fault localization in such a scenario can be achieved by minimizing the number of logs to analyze (Log-based Fault Localization)[22] or by substituting logs with the components behind them and then applying traditional Spectrum-Based Fault Localization (SBFL) to rank the components based on their suspicious scores [23]. Although the former method successfully reduces the number of logs to a small percentage, it may still result in a large number of comprehensive software systems.

 doi:10.20944/preprints202408.1096.v1

Log-Based Fault Localization (LBFL) applies a technique similar to code coverage methods but is tailored specifically for log analysis. This approach adjusts the calculation of failure and pass occurrences by normalizing repetitions within a single file. Afterward, the suspiciousness score is computed using the same methodology as in Spectrum-Based Fault Localization (SBFL). The result is a ranking of log templates ordered from the most suspicious to the least. However, in large systems, the number of logs with high suspiciousness scores is still overwhelming. To address this, we propose further refining the order of the log lines by incorporating contextual information. This is achieved through the use of an unsupervised log sequence segmentation method, Voting Experts [24], and we have previously demonstrated that this method is effective in segmentation and meets the human golden standard segmentation in open source logs [25].

The main contributions of this paper are as follows:

- Introduction of a new measure to calculate the suspiciousness of a segment.
- A method combining existing Log-based Fault Localization techniques with a new metric to further refine the localization results.
- Utilization of output from unsupervised log sequence segmentation for automated log analysis.
- Provision of an anonymized dataset from Nokia covering three real-world faults.

Section 2 describes the proposed method and datasets used for its evaluation. This is followed by the research questions and results in Section 3 and discussion in Section 4. The last section outlines directions for further research.

2. Materials and Methods

The proposed fault localization model is designed to identify the log messages related to failures in complex systems through the expansion of existing fault localization techniques. This model is useful when logs from normal execution of the system are available, and the traditional log based fault localization method produces a large number of rank 1 lines when applied to logs from a failed execution of the system. Logs, along with other parts of the code, constantly change, making it difficult for methods based on a long history to operate well. We propose a method that requires only one current set of normal logs and corresponding failed logs, generating a hierarchy of anomalous log lines, which is crucial from a fault localization point of view. As a result, our method overcomes this limitation and can be easily used for current/new software releases.

Our approach (Figure 1) consists of two stages:

- a log-based fault localization framework that identifies the most suspicious log lines in a failed log file,
- context-based ranking of rank 1 log lines, based on unsupervised log sequence segmentation.

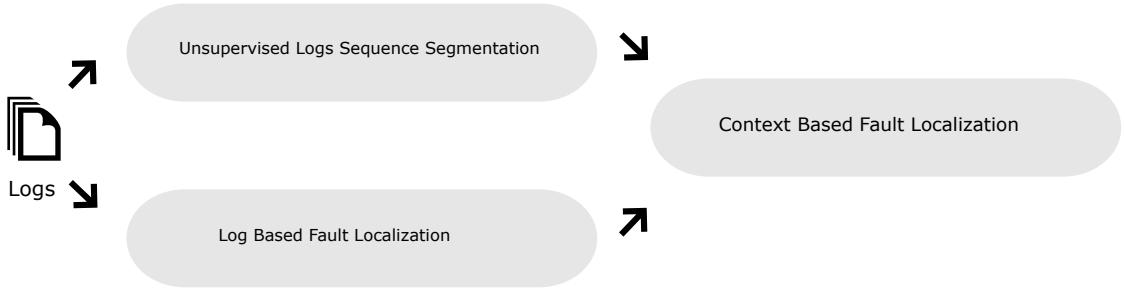

Figure 1. Overview of the approach.

 doi:10.20944/preprints202408.1096.v1

2.1. Normalization

Before analysis, the logs were normalized by removing timestamps and lines where objects were described (Figure 2). The timestamps follow a consistent format throughout the log file; therefore they do not contribute to template distinction and only add overhead to the template extraction process. We used timestamps to sort the lines, but after that, they were removed. The inclusion of object descriptions in the log makes it challenging to capture regular expressions because of variations in spaces and special punctuation marks. These objects do not contain execution-related information, and are essentially one-line information spread across multiple lines for readability. The structure and values of the printed fields of the objects can only be verified against specifications; therefore we decided to exclude these lines from consideration.

```
inf component6:   operation_type: CREATE
inf component6:   object {
inf component6:     object_name: "ObjectName1"
inf component6:     class_name: "ClassName"
inf component6:     [ClassName.object] {
inf component6:       structure1 {
inf component6:         structure2{
inf component6:           field1: "text1"
inf component6:           field1: "text2"
inf component6:           field1: 8
inf component6:           field1: 24
inf component6:           field1: 12
inf component6:           field1: 20
inf component6:           field1: 0
inf component6:           field1: 1
inf component6:           field1: 2
inf component6:           field1: 3
inf component6:           field1: 4
inf component6:           field1: 5
inf component6:           field1: 6
inf component6:       }
inf component6:     }
inf component6:   }
inf component6:   }
```

Figure 2. Description of object in log lines. Such descriptions are removed during normalization as they don't provide useful information for error localization.

2.2. Log-Based Fault localization

The normalized log contains lines of interest; however, the lines are composed of constant and variable parts. The variable parts contain information inserted in log lines during execution, and documentation is required to examine their correctness. Therefore, the next step after normalization is to remove variable elements from the lines and create so-called log templates. A well-established method for log template extraction is Drain [26], which performs this task with the highest accuracy. For example, in the messages "Status of connection to IP:192.168.11.1 is SUCCESS" and "Status of connection to IP:192.168.11.2 is FAILURE," the constant part is "Status of connection to IP:* is *," and the variable parts are ['192.168.11.1', 'SUCCESS'] and ['192.168.11.2', 'FAILURE']. This method relies heavily on the quality of the provided regular expressions, as it must correctly distinguish between the constant and variable parts of the log lines. This distinction is not trivial, and it has been shown that improvements at this stage can lead to improvements in downstream tasks [27].

Let L_n be the set of all log templates in normal logs, and L_f be the set of all log templates in the failed logs.

The suspiciousness of a log template l is defined as:

$$Susp(l) = \frac{\frac{Fail(l)}{F_T}}{\frac{Fail(l)}{F_T} + \frac{Pass(l)}{P_T}}, \tag{1}$$

where F_T, P_T are the number of all failed and passed logs, respectively, $Fail(l)$ is the number of times log template l was seen in the failed logs, and $Pass(l)$ is the number of times log template l was seen in the passed logs. These definitions differ from those of the original LBFL, where occurrences are normalized. We decided not to normalize occurrences, as we would like to operate in situations where there is only one failed, and normal logs and the frequency of log templates are important sources of information for our context-based approach. Figure 3 presents two histograms of suspicious scores; one for normalized and the other for non-normalized occurrences. Normalized occurrences are much less differentiated, whereas in our approach, frequency information is utilized to make the context more useful.

 doi:10.20944/preprints202408.1096.v1

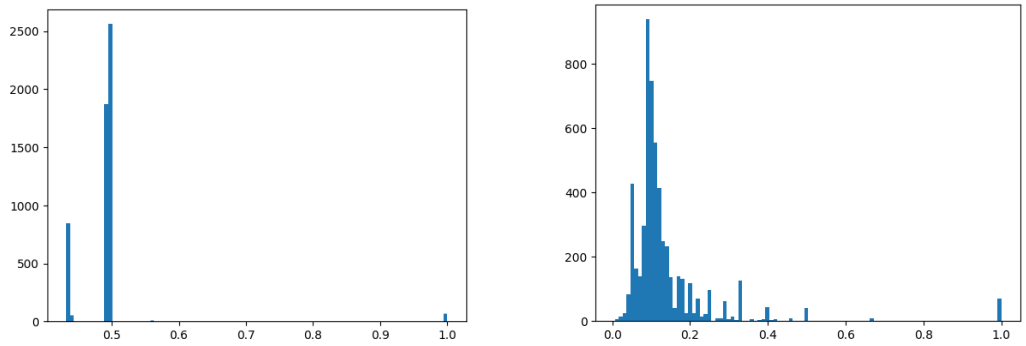

Figure 3. Occurences metrics used in (a) standard LBFL, and (b) Context-based approach.

The *Susp* metric has several important properties. It is equal to 1 for any log template that is exclusively present in failed log files and 0 for log templates that are exclusively present in the passed log files. Ranking was provided based on these values. However, this metric cannot further grade log lines that have a rank of 1. In the case of large systems, this may result in a substantial number of log lines having the same highest rank.

Tables 1–3 show the Log-based Fault Localization with suspiciousness score and ranking proposed in [22]. An example of this is a simple OK/NOK case. Tables 1 and 2 contain log examples: the first column contains log line numbers, the second is the content of the log file, the third is the ID of the extracted log template. The last one is the number of occurrences of each template ID ('-' is used when the template ID repeats).

Table 3 show $Fail(l)$ and $Pass(l)$ values, *Susp* score and rankings based on occurrences.

Then, the suspiciousness score wass calculated using Formula 1. The lines are ranked according to the suspiciousness score. As we can see in this simple example, four lines are already marked with the highest suspicious score (rank equal to 1).

Table 1. Normal log example.

#	Log content	Template ID	Pass(l)
1	INF, State change notif received [CONNECTED]	1	7
2	INF, State change notif received [CONNECTED]	1	-
3	INF, State change notif received [CONNECTED]	1	-
4	INF, State change notif received [CONNECTED]	1	-
5	INF, State change notif received [CONNECTED]	1	-
6	INF, State change notif received [CONNECTED]	1	-
7	INF, State change notif received [CONNECTED]	1	-
8	INF, SW Version: xxxxx	2	1
9	INF, currentImageType = package.01	3	1
Length	9	3	

 doi:10.20944/preprints202408.1096.v1

Table 2. Failed log example.

#	Fail log	Template ID	Fail(l)
1	WRN, BlackBoxLoggingService MemoryItem::update	4	1
2	WRN, BlackBoxLoggingService get entry value	5	1
3	WRN, BlackBoxLoggingService get entry value	5	-
4	INF, State change notif received [FAILED]	1	4
5	INF, State change notif received [CONNECTED]	1	-
6	INF, Recovery action requested	6	1
7	INF, State change notif received [CONNECTED]	1	-
8	INF, State change notif received [CONNECTED]	1	-
9	INF, Received reset request because of node2 failure.	7	1
10	INF, SW Version: xxxxx	2	1
11	INF, currentImageType = package.01	3	1
Length	11	7	

Table 3. Not normalized Log-based fault localization.

#	Template ID	Fail(l)	Pass(l)	Susp	Rank
1	1	4	7	0.36	3
2	2	1	1	0.5	2
3	3	1	1	0.5	2
4	4	1	0	1.0	1
5	5	2	0	1.0	1
6	5	1	0	1.0	1
7	5	1	0	1.0	1

2.3. Unsupervised Log Sequence Segmentation

Log sequence, as a part of the execution path, conveys some information on the software system and its execution. It contains events that are visible to humans and allows us to distinguish patterns and segments of lines as they reflect the architecture of the system. Following this observation, a few approaches have been proposed to extract this structure to ease manual examination of the log file (for example [24,28]). However, the application of these methods in automated log analysis has not yet been demonstrated. One of our contributions is that they can be used not only to improve the manual inspection of logs. We used VotingExperts [24] to extract meaningful segments from long log sequences and calculate the rankings of localized suspicious lines. Let S represent the set of all sequences of log templates

$$\mathcal{S} = \{\hat{e}^0, ..., \hat{e}^m\}, \tag{2}$$

where $\hat{e}^i$ is i-th sequence and m is the number of all sequences. The sequence is obtained by extracting all log lines belonging to one thread, block ID, or node ID. A single sequence $\hat{e}^j$ from $\mathcal{S}$ contains a sequence of log templates l_i:

$$\hat{e}^j = < l_0, ..., l_n > \tag{3}$$

Each l_i belongs to a set of log templates $\mathcal{L}$ contained in the log file. Segment $Segm_k$ is the sequence of n_k log templates from $\hat{e}^j$.

$$Segm_k = < l_{i_k}, ..., l_{i_{k+1}} > \tag{4}$$

where $i_k >= 0$ and $i_k < n$. For the sake of simplicity, for further details, please refer to the original work [24] or our previous work [25], where we showed how unsupervised word segmentation methods can be transferred to the log sequence segmentation domain. For our experiments, we used VotingExperts with window size of seven and threshold four.

 doi:10.20944/preprints202408.1096.v1

2.4. Context Based Ranking

Existing log-based fault localization methods struggle when there are many previously unseen logs. This situation is common in large software systems where a single issue can cause a cascade of failures and generate thousands of related logs. Simply returning all of them may not be helpful.

To address this issue, we propose a ranking based on context. Intuitively, unseen log lines appearing within the context of well-known log lines are more suspicious than unseen log lines grouped together. The reas is that grouped unseen log lines are often stack traces, crash dumps, or new functionalities, whereas a single anomaly midst well-known behavior can be the first symptom of a program going off the track. To calculate this metric, we first segment the log file using unsupervised log sequence segmentation, and then calculate the mean of suspicious scores of log templates in the segments where the most suspicious log templates are present. Context-based ranking was calculated using the following equation:

$$Context\_ranking(Segm_k) = 1 - \frac{\sum_{l_i \in Segm_k} Susp(L_i)}{|Segm_k|}, \tag{5}$$

where $Segm_k$ denotes the k-th segment of log file. The greater the mean value of the segment, the less suspicious are the lines.

In the example in Table 2, the fault is located in line 6. It appears in the context of successful connection responses and explains subsequent node failure. Let us assume that the first segment of the failed log is from lines 1 to 3. The sequences of the template IDs is $4, 5, 5$, and all lines were ranked as suspicious. The context ranking of this segment was calculated as $Context\_ranking(S_1) = 1 - 1 = 0$. The second segment is from lines 4 to 8 with a sequence of template IDs $1, 1, 6, 1, 1$, and the context ranking of this segment is $Context\_ranking(S_2) = 1 - mean(4 * 0.36 + 1) = 0.5$. The last segment is from lines 9 to 11, and its context ranking is $Context\_ranking(S_3) = 1 - mean(0.5 + 0.5 + 1) = 0.3$.

The segment with the highest context ranking was S_2, which was the expected value. The third segment is lower in the hierarchy. The first segment is correctly the lowest in the ranking, as it is not related to the actual fault, but represents an additional logger turned on for the expected failure.

2.5. Dataset

We performed our study on real industrial logs from Nokia, which deliver wireless connectivity solutions to many different enterprises. The reliability of communication is expected nowadays, putting pressure on the software and hardware components of the Base Transceiver Station (BTS). The software component of the BTS consists of many components that communicate over the interfaces. Failure of any component often leads to an unacceptable drop in performance. Detecting and fixing faults often can only be performed by comparison with previous normal behaviors. However, simple manual comparisons of log files consisting of hundreds of thousands of log lines are not feasible. The dataset used in this study, which is publicly available along the code [29], comprises anonymized logs from Nokia containing three faults. Anonymization was performed by removing the content of the log templates, leaving only template IDs. Thread names and level info details were also anonymized by substituting with the "thread<num>" and "level<num>" strings. For the first two fault scenarios, we collected 10 logs from normal execution prior to the fault date. For the third one, there was only one normal log and one failed log. The logs were normalized by removing timestamps and lines with interface object descriptions, segmented by thread ID, and processed using Drain. VotingExperts was then used to segment the log template sequences from each thread.

2.5.1. Example 1

In this scenario, Nokia's software failed because of the lack of communication with one of the nodes. The failed communication was logged in the middle of the normal communication with other nodes. At the beginning of the logs, there were many additional entries from the logging module,

 doi:10.20944/preprints202408.1096.v1

which had been turned on by the tester to ensure that all possible logs were collected. These logs, which are not usually enabled, contaminated the logs with false positives ranked as 1.

The failed log contained 250,578 lines with 5,557 templates, while ten normal logs contained 1,172,986 lines with 6,124 templates.

2.5.2. Example 2

In this scenario, the Nokia software failed because of a timeout on one of the mutexes. The timeout was logged during normal behavior. Previous logs with a ranking of 1 in the suspicious score were due to the newly introduced functionality, which was not the source of the problem. This issue was occasional and occurred by chance when the new functionality was turned on.

The failed log contained 90,463 lines with 4,624 templates, whereas the normal logs contained 676,942 lines with 5,475 templates.

2.5.3. Example 3

In this scenario, we had a limited number of logs, as the normal log contained only 8,188 lines and the failed log contained 6,980 logs. The Nokia software failed because of the incorrect setup of some interfaces. The real difference was in some functionality not being executed; however, this can be localized by determining the recovery actions being taken. Therefore, the most important logs were all logs containing string "RecoveryService".

The failed log contained 6,980 lines with 1,268 templates, whereas the normal logs contained 8,188 lines with 1,282 templates.

3. Results

The purpose of this work was to localize logs related to faults by reducing the number of logs an engineer has to analyze when traditional LBFL methods return many log lines with the highest rank of 1. The data used were labeled by experts to identify the crucial log in each scenario that revealed the real root cause. We collected the number of segments sorted by context ranking and standard timestamp. We consider our method better if it ranks the most important log template with a lower rank than the timestamp approach, and if going from the top of the ranking to the obtained segment, the developer has to examine a lower number of log templates. Suspicious log lines are presented to the developer in an original form, not in a log template form, as it is easier to analyze and understand. It is thus preferable to group all instances of the log template together so that after seeing one instance from the group, the developer may skip the rest. To measure the gain we use a percentage of reduction of data to analyze, calculated from the equation:

$$Red = \left(\frac{X_1 - X_2}{X_1} \right) \times 100 \tag{6}$$

, where X_1 represents the baseline value, and X_2 represents the obtained value. During our experiments, we aimed to answer two research questions:

RQ1: Is segment ranking by context suspicious score more effective than the standard timestamp approach?

RQ2: Does ranking by context suspicious score reduce the number of distinct log templates to check?

RQ1: *Is segment ranking by context suspicious score more effective than the standard timestamp approach?*

In Example 1, after the application of the traditional LBFL, there were 70 log templates with a suspicious score of 1, and 105 segments obtained by VotingExperts contained these log templates. The most crucial log template was 5992, located in segment 64 when sorted by timestamps. Our method ranked the most important line in segment 22. With segments sorted by timestamp, the most important log template was in the 64th segment. The number of segments to be analyzed was reduced by 65.6%. In Example 2, the LBFL marked 665 log templates with a suspicious score of 1, and 1,629 segments in

 doi:10.20944/preprints202408.1096.v1

the failed log contained these log templates. The most crucial log template was template 18 in segment 736, when sorted by timestamp. The log snippet with failure from this example is shown in Figure 4. Context-based ranking ranked the most important line in position 303, compared to the 737th segment in the timestamp approach. The number of segments to be analyzed was reduced by 59%. In Example 3, LBFL returned 86 log templates with a suspicious score of 1, and 319 segments from VotingExperts contained those log templates. The most crucial log templates were 16, 17, and 1297. Context-based ranking ranked the most important segment at position 34, compared with the 44th segment in the timestamp approach. The number of segments to be analyzed was reduced by 22.7%.

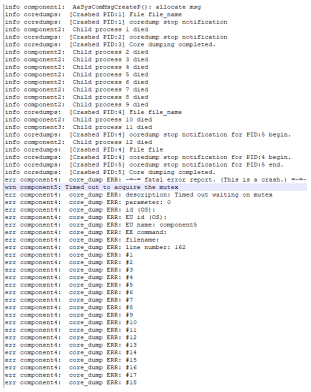

Figure 4. Log snippet from Example 2 with the most important line being marked.

RQ2 *Does ranking by context suspicious score reduce the number of distinct log templates to check?*

Example 1, with context-based ranking segments from 1 to 22nd, contained 18 different log templates for verification. In the timestamp ranking, 38 different log templates were visible in segments 1-64. The reduction in the number of distinct templates for these analysis was 52.6%. In Example 2, with context-based ranking in the first 303 segments, 104 different log templates were for verification. With timestamp ranking, 289 templates were contained in the 737 first segments. The reduction in the number of distinct templates for these analysis was 64%. In Example 3, with context-based ranking, 17 different log templates were to be verified in the first 34 segments. Using timestamp ranking, 23 templates were contained in the 44 first segments. The reduction in the number of distinct templates for these analysis was 26.1%.

In all cases, our solution outperformed the standard timestamp approach, providing the possibility of identifying the most important log lines more quickly, as presented in Table 4.

Table 4. Comparison of Context-Based Ranking and Standard Timestamp Approach.

Ex.	Approach	Rank	Templ. No.	Seg. Red. (%)	Templ. Red. (%)
1	Standard Timestamp	64	38	-	-
1	Context Suspicious Score	22	18	65.6	52.6
2	Standard Timestamp	736	289	-	-
2	Context Suspicious Score	302	104	59.0	64.0
3	Standard Timestamp	44	23	-	-
3	Context Suspicious Score	34	17	22.7	26.1

4. Discussion

The proposed Log-based Fault Localization with context-based ranking for rank 1 lines proved to be more effective than timestamp ranking in the provided fault localization scenarios. Context-based ranking reduces the number of segments to be analyzed as well as the number of distinct log templates. The former implies that developers must analyze fewer logs to encounter the most important one. The latter implies that log lines with the same log template are grouped together; therefore after seeing one such example, a developer can skip subsequent ones. In contrast, the timestamp approach forces the developer to analyze many more log templates and their contexts before reaching the crucial log line.

 doi:10.20944/preprints202408.1096.v1

This demonstrates that log-based methods can be applied in situations with large differences in log templates between normal and failed logs, where traditional LBFL returns a substantial portion of the same rank 1 log templates for analysis, and context can be used to refine the analysis.

Our approach uses the context of log templates obtained from unsupervised log sequence segmentation, showing that the segmentation of log sequences can be useful for downstream tasks, such as fault localization.

For further research, it would be interesting to investigate how the quality of the Drain influences the results. Because Drain is the main source of information regarding log templates and is highly dependent on the quality and accuracy of regular expressions, it would also be valuable to experiment by substituting this method with neural-based semantic approaches that are less reliant on human effort. Further experiments can also be done using sophisticated neural-based methods to compare segments, including Transformer based methods.

Author Contributions: Conceptualization, W.D.; methodology, W.D. and O.U.; software, W.D. and K.I.K.; validation, W.D., K.I.K.; formal analysis, O.U.; investigation, W.D.; resources, O.U.; data curation, W.D.; writing—original draft preparation, W.D.; writing—review and editing, M.N. and W.D.; visualization, W.D.; supervision, O.U.; project administration, W.D.; funding acquisition, 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: Data available in a publicly accessible repository https://github.com/dobrowol/log_based_fault_localization.

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

Abbreviations

The following abbreviations are used in this manuscript:

MDPI	Multidisciplinary Digital Publishing Institute
SBFL	Spectrum-Based Fault Localization
KPI	Key Performance Indicators
FSLF	Fault Localization Statement Frequency

References

1. Zhang, S.; Xia, S.; Fan, W.; Shi, B.; Xiong, X.; Zhong, Z.; Ma, M.; Sun, Y.; Pei, D. Failure Diagnosis in Microservice Systems: A Comprehensive Survey and Analysis. *arXiv preprint arXiv:2407.01710* **2024**.
2. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F.; Li, D. Software fault localization: An overview of research, techniques, and tools. *Handbook of Software Fault Localization: Foundations and Advances* **2023**, pp. 1–117.
3. OpenTelemetry Contributors. OpenTelemetry. https://opentelemetry.io, 2024. Accessed: 2024-07-12.
4. Remil, Y.; Bendimerad, A.; Mathonat, R.; Kaytoue, M. Aiops solutions for incident management: Technical guidelines and a comprehensive literature review. *arXiv preprint arXiv:2404.01363* **2024**.
5. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F.; Li, D. Software fault localization: An overview of research, techniques, and tools. *Handbook of Software Fault Localization: Foundations and Advances* **2023**, pp. 1–117.
6. Jones, J.A.; Harrold, M.J.; Stasko, J. Visualization of test information to assist fault localization. Proceedings of the 24th international conference on Software engineering, 2002, pp. 467–477.
7. Wong, W.E.; Debroy, V.; Gao, R.; Li, Y. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* **2013**, *63*, 290–308.
8. Abreu, R.; Zoeteweij, P.; Van Gemund, A.J. On the accuracy of spectrum-based fault localization. Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007, pp. 89–98.
9. Wong, W.E.; Qi, Y. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* **2009**, *19*, 573–597.
10. Li, Y.; Wang, S.; Nguyen, T. Fault localization with code coverage representation learning. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 661–673.

 doi:10.20944/preprints202408.1096.v1

11. Wardat, M.; Le, W.; Rajan, H. Deeplocalize: Fault localization for deep neural networks. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 251–262.
12. Zhang, Z.; Lei, Y.; Mao, X.; Li, P. CNN-FL: An effective approach for localizing faults using convolutional neural networks. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 445–455.
13. Weiser, M.D. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*; University of Michigan, 1979.
14. Binkley, D.; Gold, N.; Harman, M. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2007**, *16*, 8–es.
15. Agrawal, H.; Horgan, J.R. Dynamic program slicing. *ACM SIGPlan Notices* **1990**, *25*, 246–256.
16. Zhang, X.; Gupta, N.; Gupta, R. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering* **2007**, *12*, 143–160.
17. Alves, E.; Gligoric, M.; Jagannath, V.; d'Amorim, M. Fault-localization using dynamic slicing and change impact analysis. 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011, pp. 520–523.
18. Mao, X.; Lei, Y.; Dai, Z.; Qi, Y.; Wang, C. Slice-based statistical fault localization. *Journal of Systems and Software* **2014**, *89*, 51–62.
19. Zhu, J.; He, P.; Fu, Q.; Zhang, H.; Lyu, M.R.; Zhang, D. Learning to log: Helping developers make informed logging decisions. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, Vol. 1, pp. 415–425.
20. Xu, J.; Chen, P.; Yang, L.; Meng, F.; Wang, P. Logdc: Problem diagnosis for declartively-deployed cloud applications with log. 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE). IEEE, 2017, pp. 282–287.
21. Zhang, Q.; Jia, T.; Wu, Z.; Wu, Q.; Jia, L.; Li, D.; Tao, Y.; Xiao, Y. Fault localization for microservice applications with system logs and monitoring metrics. 2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA). IEEE, 2022, pp. 149–154.
22. Sha, Y.; Nagura, M.; Takada, S. Fault localization in server-side applications using spectrum-based fault localization. 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2022, pp. 1139–1146.
23. de Castro Silva, J.M.R.; others. Spectrum-Based Fault Localization for Microservices via Log Analysis **2022**.
24. Cohen, P.; Heeringa, B.; Adams, N.M. An unsupervised algorithm for segmenting categorical timeseries into episodes. Pattern Detection and Discovery: ESF Exploratory Workshop London, UK, September 16–19, 2002 Proceedings. Springer, 2002, pp. 49–62.
25. Dobrowolski, W.; Libura, M.; Nikodem, M.; Unold, O. Unsupervised Log Sequence Segmentation. *IEEE Access* **2024**, pp. 1–1. doi:10.1109/ACCESS.2024.3409425.
26. He, P.; Zhu, J.; Zheng, Z.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. 2017 IEEE international conference on web services (ICWS). IEEE, 2017, pp. 33–40.
27. Wu, X.; Li, H.; Khomh, F. On the effectiveness of log representation for log-based anomaly detection. *Empirical Software Engineering* **2023**, *28*, 137.
28. Shani, G.; Meek, C.; Gunawardana, A. Hierarchical probabilistic segmentation of discrete events. 2009 Ninth IEEE International Conference on Data Mining. IEEE, 2009, pp. 974–979.
29. Dobrowolski, W. Context Based Fault Locaclization. https://github.com/dobrowol/log_based_fault_localization, 2024. Accessed: 12-Aug-2024.