2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-1594-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSRE59848.2023.00018

EvLog: Identifying Anomalous Logs over Software Evolution

Yintong Huo[†], Cheryl Lee[†], Yuxin Su[‡*], Shiwen Shan[‡], Jinyang Liu[†], and Michael R. Lyu[†]
[†]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China.
Email: {ythuo, jyliu, lyu}@cse.cuhk.edu.hk, cheryllee@link.cuhk.edu.hk
[‡]Sun Yat-sen University, Zhuhai, China. Email: suyx35@mail.sysu.edu.cn, shanshw@mail2.sysu.edu.cn

***Abstract*—Software logs record system activities, aiding maintainers in identifying the underlying causes for failures and enabling prompt mitigation actions. However, maintainers need to inspect a large volume of daily logs to identify the anomalous logs that reveal failure details for further diagnosis. Thus, how to automatically distinguish these anomalous logs from normal logs becomes a critical problem. Existing approaches alleviate the burden on software maintainers, but they are built upon an improper yet critical assumption: logging statements in the software remain unchanged. While software keeps evolving, our empirical study finds that evolving software brings three challenges: log parsing errors, evolving log events, and unstable log sequences. In this paper, we propose a novel unsupervised approach named Evolving Log analyzer (EvLog) to mitigate these challenges. We first build a multi-level representation extractor to process logs without parsing to prevent errors from the parser. The multi-level representations preserve the essential semantics of logs while leaving out insignificant changes in evolving events. EvLog then implements an anomaly discriminator with an attention mechanism to identify the anomalous logs and avoid the issue brought by the unstable sequence. EvLog has shown effectiveness in two real-world system evolution log datasets with an average F1 score of 0.955 and 0.847 in the intra-version setting and inter-version setting, respectively, which outperforms other state-of-the-art approaches by a wide margin. To our best knowledge, this is the first study on localizing anomalous logs over software evolution. We believe our work sheds new light on the impact of software evolution with the corresponding solutions for the log analysis community.**

I. Introduction

Nowadays, intelligent log analytics is designed to manage overwhelming logs [1] for failure troubleshooting, and anomaly detection [2]. Existing automated log analytics can be categorized into two types based on granularity: coarse-grained tasks and fine-grained tasks. Coarse-grained models, such as the anomaly detectors [3] and failure predictors [4], detect (or predict) anomalies given the logs from a period of time. Taking anomaly detection as an example, the model accepts a session of logs to determine whether an anomaly exists in this session. Although the coarse-grained models show promising results in open datasets, they provide limited evidence of failure diagnosis for software maintainers. On the other hand, fine-grained tasks aim to further identify the individual/single anomalous logs within a session showing possible interpretations of the failure [5], [6], [7]. Even if coarse-grained models free maintainers from inspecting massive log lines, it is still time-consuming to analyze hundreds of log lines within a session to find the anomalous log for troubleshooting [8]. To ease the burden of software maintainers, we focus on this more challenging yet significant task, individual anomalous log identification, in this paper.

An *anomalous log* signals an anomaly in the system, such as network error [8]. The following example shows a log message that may indicate a connection problem caused by a network fault within the system:

```
Container launch failed for container_32h: Connection refused.
```

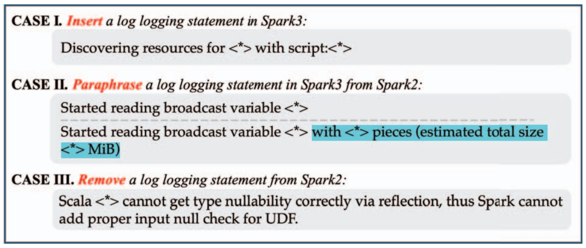
Anomalous logs are crucial for diagnosing failures, but they are often accompanied by numerous normal logs, which can be overwhelming for maintainers. To distinguish them from normal run-time logs, existing studies [6], [7], [9], [10] constructed a *reference model* from training log sequences and then identified which log violated the reference model. Specifically, they abstracted log event sequences into a directed graph via either a finite state machine (FSM) [6], [7], [9] or causal dependencies [10] as the reference model. Subsequently, any deviations from this model would be regarded as an indication for anomaly and marked for troubleshooting.

However, both FSM-based and causal graph-based approaches following the *closed-world assumption* suffer limitations for processing the unseen data. However, after the initial version is released, software experiences continual development to fulfill customers' demand, to fix bugs, and to extend to new functionalities, which is well-known as *software evolution* [11], [12]. Previous studies pointed out that logging statements change over software evolution is so pervasive that around 33% of the log are revised as after-thoughts [13], [14]. The changed logging statements during the evolution activities raise challenges for existing approaches:

(1) Parsing errors. Log parsers extract static events (e.g., *Container launch failed for <*>: Connection refused.*) and dynamic parameters (e.g., *container_32h*) from log messages. However, as discussed in Section II-B1, parsers may misalign revised log events in evolving software versions, causing log parsing errors. These parsing errors further downgrade the subsequent log analytics performance. ***(2) Evolving events.*** Even if state-of-the-art parsers work as expected, software evolution brings new logging statements or paraphrases old logging statements, which we refer as *evolving events* in this paper. ***(3) Unstable sequences.*** Apart from log events, the log

[*] Corresponding author.

2332-6549/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSRE59848.2023.00018



CASE I. *Insert a log logging statement in Spark3:*

Discovering resources for <*> with script:<*>

CASE II. *Paraphrase a log logging statement in Spark3 from Spark2:*

Started reading broadcast variable <*>

Started reading broadcast variable <*> with <*> pieces (estimated total size <*> MiB)

CASE III. *Remove a log logging statement from Spark2:*

Scala <*> cannot get type nullability correctly via reflection, thus Spark cannot add proper input null check for UDF.

Fig. 1. Evolving logging statement cases for Spark2 and Spark3.

sequences from running identical jobs can vary, named *unstable sequences*. Such variation can be caused by interleaving logs produced from multiple threads [6]. Moreover, software evolution may alter the function invocation sequences, leading to new sequential patterns.

While solutions to the first two challenges still remain unexplored, there have been several attempts to handle the third challenge. For example, previous study [15] tried to resolve the interleaving logs by considering multiple predecessors and successors of a log event, instead of just the direct ones. Another study [16] mitigated unstable sequences challenge by learning causal relationships between event pairs from historical data. Nevertheless, none of the existing approaches considered the software evolution scenario, which can negatively impact the performance of identifying anomalous logs if left unaddressed.

To address the above challenges, we propose an unsupervised anomalous log identification solution over software evolution. The design of our approach is based on two insights: *1) the majority of logs are normal in a healthy system; and 2) the anomalous logs are unknown a priori because we cannot iteratively inject all kinds of failures.* In particular, we design EvLog with two steps. The first step aims to tackle the *parsing errors* and the *evolving events* issues. We derive multi-level representations directly from logs to prevent introducing *parsing errors*. The representations at different levels undertake different functions: 1) the semantic-rich representation aims to fully retain semantics from log messages, which is extracted by pre-trained language models; and 2) the abstract representation to align similar logs across software evolution, which is derived from the hierarchical clustering approach. Such multi-level representations maintain the pertinent semantics while leaving out unnecessary trifles to address the *evolving events* issue.

In the second step, we address the *unstable sequence* issue by constructing an anomaly discriminator with an attention mechanism. The core idea is to learn a transformation function (e.g., neural networks) that embeds normal log features (source domain) to stay close (enclosed in a hyper-sphere) to a target domain, then the logs that are largely distant from this hyper-sphere are considered as anomalous ones. Specifically, EvLog constructs log features for each single log and its surrounding log contexts based on multi-level representations. It then applies neural networks to discriminate the anomalous logs instead of rigorously comparing new sequences with existing ones. Once trained, EvLog can be directly applied to a future software version without any fine-tuning.

TABLE I
LOGGING EVOLUTION RATIO BETWEEN SPARK2 AND SPARK3.

Percentage	Unchanged	Inserted	Paraphrased	Removed
Log message	91.16%	0.07%	8.75%	0.02%
Logging statement	76.12%	12.69%	1.49%	9.70%

Our new approach is evaluated using two realistic datasets (i.e., LOGEVOL) and a synthetic dataset (i.e., SYNEVOL) to simulate logging evolution. The experiment results illustrate that EvLog reaches a promising average F1 score of 0.955 and 0.847 in intra-version identification and inter-version anomalous log identification on two representative system logs, respectively.

To conclude, the contribution of this paper is threefold:

- We empirically identify three challenges (i.e., parsing errors, evolving events, unstable sequence) brought by software evolution for anomalous log identification, which has never been properly addressed before.
- To overcome the above challenges, we develop EvLog, an unsupervised anomalous log identification approach with a multi-level representation extractor and an anomaly discriminator. To our best knowledge, EvLog is the first solution to tackle the problem of identifying anomalous logs over software evolution.
- By evaluating EvLog on real system log datasets and a synthetic dataset, we show our approach can effectively identify anomalous logs across different software versions without fine-tuning or manual labeling. Artifacts are released for research purposes at https://github.com/YintongHuo/EvLog.

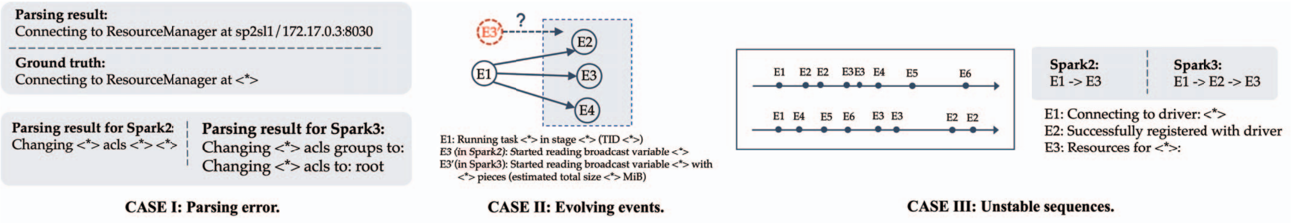
II. MOTIVATING STUDY

A. How do logging statements evolve?

Developers may modify logging statements when updating the software, producing unseen log messages in system runtime for maintenance. To examine how logging statements evolve during software updates, we analyze Spark, an open-source cluster computing system for the parallel processing of large-scale data. In particular, we run benchmark workloads in Spark 2.4.0 (denoted as Spark2) and Spark 3.0.3 (denoted as Spark3) with details shown in Section V-A and compare the collected log messages.

We categorize the change of logging statements into three types: *insert*, *paraphrase* and *remove*. We show three cases in Fig. 1, where "<*>" refers to the dynamic parameters generated in running time. In Case I, a new logging statement is added in Spark3 to indicate the attached resources. In Case II, the logging statement is paraphrased by adding information on the number of pieces and the estimated size of the variable to gain a deeper understanding of the system performance. In Case III, a logging statement is removed from Spark2 due to the deprecation of "*UserDefinedFunction*" in Spark3.

Table I displays the statistics of the three types of changes on collected log messages and logging statements. It is observed that nearly 24% logging statements are changed from Spark2 to Spark3, resulting in almost 10% changed log



```
Connecting to ResourceManager at sp2sl1/172.17.0.3:8030
Connecting to ResourceManager at <*>
Changing <*> acls <*> <*>
Changing <*> acls groups to:
Changing <*> acls to: root
```

E1: Running task <*> in stage <*> (TID <*>)
E3 (in Spark2): Started reading broadcast variable <*>
E3'(in Spark3): Started reading broadcast variable <*> with <*> pieces (estimated total size <*> MiB)

E1: Connecting to driver: <*>
E2: Successfully registered with driver
E3: Resources for <*>:

CASE I: Parsing error.

CASE II: Evolving events.

CASE III: Unstable sequences.

Fig. 2. Three challenges brought by software evolution. E1, E2, etc., represent different log events. (1) The parsing error case shows the incorrectly parsed log messages that will impact the subsequent log analysis, (2) The evolving event case exhibits that a paraphrased logging statement will mislead the reference model, and (3) The unstable sequence case depicts how a new logging statement E2 can alter produced log sequences.

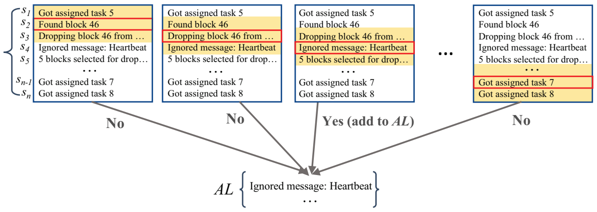
messages. Although 12.69% and 9.70% logging statements are inserted or removed, respectively, they only make up less than 0.1% collected logs, meaning they appear in a low frequency. However, the high proportion of paraphrased logs implies developers are likely to modify the commonly-used logging statements. To conclude, logging statements change over software evolution. The non-negligible amount of changes motivates us to reckon with the software evolution issue.

B. How does evolution raise challenges for anomalous log identification approaches?

1) Parsing errors: Log parsers extract constant strings (i.e., events) and run-time parameters from log messages. However, existing log identification models *only* use the extracted events and do not consider the original log messages. This can be problematic because log parsers can introduce errors, and the evolution of logs over time can make parsing even more challenging [17]. CASE I in Fig. 2 displays two parsing mistakes from a widely-used parser, Drain [18], where <*> denotes parameters. The top one is caused by confusing parameters with constant strings, and the bottom one shows inconsistent parsing results in Spark2 and Spark3. Since current log parsers are parameter-sensitive and not versatile enough [19], and the hyper-parameters that work well for one software version may not be suitable for others. Since systematic log analytics should operate on raw log messages, it is essential to find ways to avoid parsing mistakes.

2) Evolving events: Log identification models also face a challenge in dealing with evolving events. Typically, these models detect anomalous logs by examining whether the actual next log is in line with the predicted next logs based on contextual information. The idea works well when all the events are known; however, if the actual next log is an unseen event, it can never be matched with any predicted next logs. For instance, in CASE II of Fig. 2, event E3' cannot be matched with the predicted logs since it is a paraphrase of event E3. According to its decision logic, such inconsistency leads to a significant issue where all unseen events are treated as false positives. This issue becomes severe for all existing log-based approaches considering that 8.75% of the collected logs have been paraphrased.

3) Unstable sequences: Ideally, we expect that the log message sequences perfectly match the execution sequences of a program. However, there are situations where log messages from different threads can interleave, resulting in what we refer to as ”unstable sequences”. Additionally, introducing new logging statements in a software update can create new log events during run-time, leading to sequential pattern changes. As shown in CASE III of Figure 2, unstable sequences can be caused by interleaving logs [6] and new log events from software evolution. To resolve the issue, identifying relevant and informative log messages in a sequence is of great essence.



```
Got assigned task 5
Found block 46
Dropping block 46 from …
Ignored message: Heartbeat
5 blocks selected for drop…
…
Got assigned task 7
Got assigned task 8
```

```
Ignored message: Heartbeat
…
```

Fig. 3. Anomalous logs localization problem illustration.

In summary, our empirical findings suggest that logging evolution can affect existing models in three ways: the potential parsing errors, the evolving events, as well as the unstable sequences. These influential factors have never been explored, yet their impact can hardly be ignored.

III. Problem Illustration

In this paper, we consider the anomalous log identification problem as in the literature [6], [7], [20], which enables pinpoint a collection of fault-indicating anomalous logs [8]. Given a sequence of log messages $s = s_1, s_2, ..., s_n$, the task asks the model to find a set of anomalous logs $AL = \{s_i | 1 \leq i \leq n\}$ within the message sequence. Compared with the anomaly detection task that determines whether a problem exists in a session (session-based), anomalous log identification is a more fine-grained and challenging task that needs to localize individual fault-indicating logs (message-based). We use *context* to represent the surrounding logs of a specific log (named as the *center log*) and analyze whether the log is anomalous based on its context.

To resolve the subset AL, we check every individual log, that is, for all $s_i \in s$, the model determines whether the center log s_i is an anomalous log in the given context of s_i. We will add the center log into AL if it is considered anomalous. Fig. 3 shows the identification process, where the center log and its corresponding context are highlighted with a red rectangular and yellow background, respectively.

IV. Approach

This section introduces our novel approach, called EvLog, shown in Fig. 4, in tackling the anomalous log identification

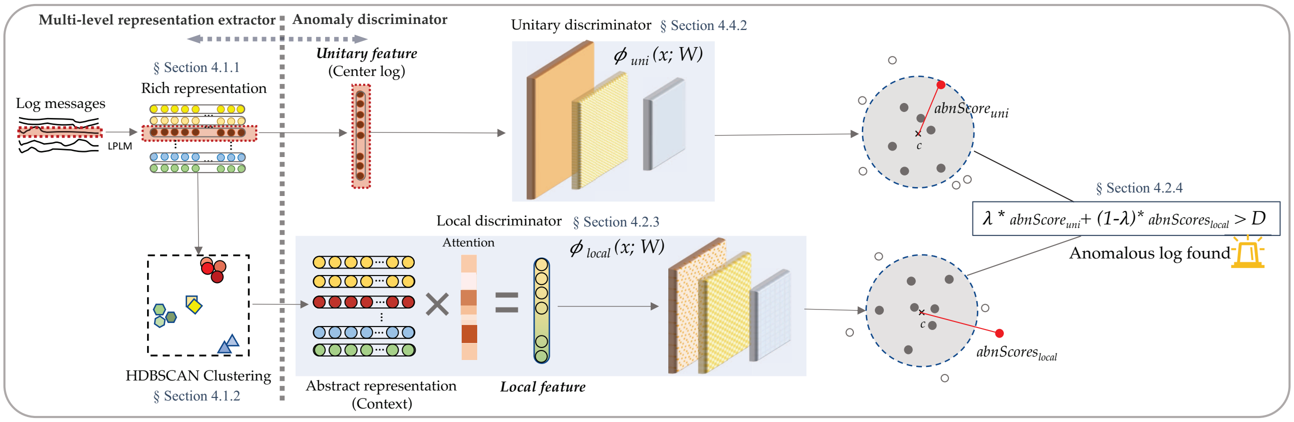


Fig. 4. EvLog with a multi-level representation extractor and an anomaly discriminator.

challenges over software evolution. EvLog has two components, i.e., a *multi-level representation extractor* to derive multi-level robust log representations, followed by an *anomaly discriminator* with the attention mechanism to pinpoint the anomalous logs. In particular, the multi-level representation extractor targets at extracting *rich representations* as informative as possible and *abstract representations* to capture high-level commonalities among similar logs. Then these representations are fed into the anomaly discriminator to automatically localize the anomalous logs in an unsupervised manner.

A. Multi-level representation extractor

We exploit multi-level representations with various information from log messages to semantically understand them. This section illustrates how to extract multi-level semantic representations, that is, a *rich representation* and an *abstract representation*. The low-level rich representation provides a concrete understanding of a certain log. In contrast, the high-level abstract representation captures the commonality of logs with similar semantics, regardless of their slight differences (e.g., parameters difference, revised log events).

1) Rich representation: Semantics in both log events and their corresponding parameters has advantageous for log analysis [21], [22]. To obtain informative representations from logs with respect to their semantics, we fine-tune a pre-trained language model [23] (PLM) on our collected log datasets.

The PLMs have shown the powerful semantic encoding ability for many software engineering tasks, such as log-based anomaly detection [24] and code comprehension [25]. In our work, to overcome potential parsing errors and to make the best usage of information inside log messages, EvLog acquires domain-specific semantic representations via PLMs. On one side, system logs share some fundamental knowledge with natural languages since humans write logging statements. After being trained on a large corpus, the PLMs learn more information about word senses, not limited to system logs. On the other side, we notice that these PLMs are not sufficient for domain-specific tasks due to the knowledge gap. Hence, we fine-tune the massive language models to further capture domain-specific semantics. In specific, we employ the widely-used masked language modeling strategy [26], [27], [28] to fine-tune the PLMs, by randomly masking 10% tokens in each log and asking the model to predict the masked tokens.

Specifically, given a log message x, the rich representation x_{rich} is designed to capture its detailed semantics. This parser-free representation extractor accepts log messages instead of events, allowing it to get away from potential parsing mistakes.

Algorithm 1 Abstract representation acquisition.

```
Input: Rich representation to be clustered $E = [e_1, e_2, ..., e_n]$
Output: Abstract representation $C = [c_1, c_2, ..., c_n]$
$C = []$
Centroid={}
$E'$ = PRINCIPALCOMPONENTANALYSIS($E$)
ClusterIds = HDBSCAN($E'$)
"""Compute centroid for each cluster"""
for all ClusterID from 1 → SET(ClusterIds) do
    Centroid[ClusterID] = MEAN(E'[ClusterIds==ClusterID])
end for
"""Compute abstract representation"""
for all $i$ from 1 → n do
    $C$.APPEND(Centroid[ClusterIds[$i$]])
end for
```

2) Abstract representation: Apart from the rich representation, we also extract a high-level semantic representation, x_{abs}, that remains stable on similar log events over logging evolution. To this end, we develop a cluster-based approach on top of the rich representations. Previous studies [29], [30] have demonstrated the effectiveness of clustering approaches in grouping similar texts together based on their intrinsic characteristics. Motivated by theirs, we also employ a cluster-based approach to group log messages. Specifically, we adopt the idea from the previous log clustering study [31] using Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [32], whose efficiency and effectiveness has been presented in many domains [33], [34]. Compared to other clustering approaches, HDBSCAN inherits two special advantages for our scenario: (1) It can automatically extract the "dense" cluster without pre-defining the number of clusters (e.g., Kmeans [35]), which is important in the case that we may never know the number of clusters of logs. (2) HDBSCAN has a few parameter numbers, and its robustness to parameter choice [31], [32] makes it versatile for diverse log data.

Eventually, the abstract representation x_{abs} for each log message x is the centroid of its corresponding cluster by averaging all points (logs) belonging to the cluster. Algorithm 1 shows the abstract representation computation process.

B. Anomaly discriminator

This section illustrates how we pinpoint anomalous logs from the acquired rich and abstract representations. In particular, for each input log, the unitary discriminator processes the log, and the local discriminator processes the log's context. Two processed results are integrated as the final output.

1) Basic idea: Existing unsupervised log-based reference sequences models [3], [6], [36] build a reference model from training data and check whether the testing log violates the prediction from the model. Unfortunately, these models are ineffective at handling evolving events over software evolution. Moreover, the anomalous logs are unknown a priori because we cannot iteratively inject all types of faults. Thus, we propose a different approach to handle the problem by learning the "normality" of the normal log features instead of predicting the subsequent events.

Motivated by the Support Vector Machine [37] (SVM) that learns a hyperplane to separate data, our idea is to develop a neural network that learns a hyper-sphere to separate normal logs and anomalous logs. The neural network maps log features (in the source domain) to a target domain where normal features stay as close as possible (enclosed in a hyper-sphere). We measure the distance between a mapped log feature and the center of the hyper-sphere as *normality* (e.g., grey circle in Figure 4), with logs far from the center being considered anomalous due to deviating from the normality. In this way, logs with evolving events can be transformed into the target domain where they are close to the previous semantically similar logs, minimizing adverse effects on the anomaly discriminator results. For example, if a normal log is paraphrased during software evolution, the evolved log with similar semantics will be mapped within the normality. This new approach in localizing anomalous logs is superior via two advantages: 1) It delivers better performances than other traditional methods due to the neural network's proven learning ability. 2) Our approach frees humans from labor-intensive labeling since it can learn the normality naturally in an unsupervised manner from large-scale normal logs that can be easily collected from stable software.

Specifically, the goal is to train a neural network model (mapped features from the source domain to the target domain) while minimizing the hyper-sphere volume that encloses the normal data features in the target domain. In this way, the model is forced to learn implicit semantics since it must map the normal log features closely to the hyper-sphere's center. Thus the unseen log events with similar semantics can also be embedded close in the target domain. To achieve the above goal, the objective function is:

$$J = \min_{W} \quad \frac{1}{n}\sum_{i=1}^{n} \|\phi(x_i; W) - c\|^2 + \frac{\alpha}{2}\|W\|^2, \tag{1}$$

where $\phi(x_i; W)$ refers to using the model ϕ with its parameters W to map each input sample $x_i \in x$ to a hyperspace $\mathbb{R}^n$; $c \in \mathbb{R}^n$ refers to the hyper-sphere center; the last term serves as a regularization term with weight α to avoid over-fitting. The objective function forces the normal data features to stay close to the center c. Theoretically, the mapping model ϕ can be replaced by any neural network architecture, demonstrating the extensibility of our approach. The following two sections show how we develop an appropriate neural network for mapping multiple features. We then describe how to integrate the mapped features for identifying anomalous logs in Section IV-B4.

2) Unitary discriminator: We first look into single logs, as the single log that contains negative words (e.g., "failure" and "error") usually indicates an anomaly. The unitary discriminator works on rich representation of individual logs, aiming to map normal logs to a hyper-sphere that describes the normality. The motivation behind the unitary discriminator is that, the negative terms in anomalous logs exhibit significantly different semantics than words in normal logs (e.g., "running", "success"). These anomalous logs' features will be mapped far away from the center of the hyper-sphere; thus they are considered as normality-deviating ones. To this end, we adopt the strong learning ability from neural networks and build the unitary discriminator (ϕ_{uni}) with a two-layer feed-forward neural network denoted as $FFNN$. We describe the architecture as follows:

$$\phi_{uni}(x_{rich}; W_{uni}) = FFNN_b((FFNN_a(x_{rich})), \tag{2}$$

where x_{rich} refers to the rich representation containing full log semantics (i.e., *unitary feature*) of the center log x.

3) Local discriminator: Looking into one individual log is not sufficient to comprehensively understand the running status, so it is noteworthy to exploit its contextual information. On the one hand, it is pointed out that different logs possess different importance [38]. For example, some miscellaneous logs regularly appear regardless of what job the system is running, whereas other logs provide richer guidance for analysis. On the other hand, log data transmission, collection, and software evolution affect synchronization temporally, leading to unstable sequences. To focus on beneficial logs and leave the uninformative logs out, we leverage the attention mechanism [39] to focus on beneficial logs. In the local discriminator, we use the center log and its contexts to acquire a *local feature* against unstable sequences and then learn the normality of such a local feature.

Given a center log x, we construct its context representation x_{ctx} by forming its abstract representation of context as a matrix. Then we compute the weights across the context by the attention mechanism [40], allowing the model to learn the importance of surrounding logs, thus addressing the unstable sequence issue. Specifically, given a center log x_{rich} as query and its context x_{ctx} as value, we compute the weighted context representation as the local feature (denoted as x_{local}) as follows:

$$\begin{aligned} x_{local} &= softmax(\frac{x_{query}x_{ctx}^T}{\sqrt{d_k}})x_{ctx}, \\ x_{query} &= FFNN_c(x_{rich}), \end{aligned} \tag{3}$$

where d_k refers to the dimension of x_{ctx}, and $FFNN_c$ transforms x_{rich} to the target domain that shares the same dimension with x_{ctx}.

After that, another two-layer neural network with an activation function is applied to the local feature x_{local} for learning normality from contexts. To sum up, we describe the network for the local discriminator (ϕ_{local}) as in Equation 4:

$$\phi_{local}(x_{rich}, x_{ctx}; W_{local}) = FFNN_e((FFNN_d(x_{local}))). \tag{4}$$

4) Integration: The unitary discriminator learns normality for individual logs, whereas the local discriminator learns the context normality in running status. To fully exploit these two different information sources, we propose the total objective function with a weighted sum in Equation 5 to simultaneously optimize two sub-discriminators:

$$J_{total} = \lambda * J_{uni} + (1 - \lambda) * J_{local}, \tag{5}$$

where J_{uni} and J_{local} are the functions defined in Equation 1 for unitary discriminator ϕ_{uni} and local discriminator ϕ_{local}, respectively. The objective functions allow two discriminators to learn the normality by minimizing their hyper-sphere volume.

The distance between a log message (after mapping by discriminators) to the hyper-sphere center measures the degree of normality of the log. We apply an *abnormal score* ($abnScore$) to describe how the log deviates from the normality, which is the weighted sum of the abnormal sub-scores from two independent discriminators. The abnormal sub-score $abnScore_i$ is defined by the Euclidean distance from the feature embedding to its corresponding hyper-sphere center, denoted by Equation 6:

$$\begin{aligned} &abnScore = \lambda * abnScore_{uni} + (1 - \lambda) * abnScore_{local}, \\ &abnScore_i = \|\phi_i(x; W_i) - c_i\|^2, i \in \{uni, local\}. \end{aligned} \tag{6}$$

The center log is eventually predicted as an anomaly if and only if its abnormal score is larger than the threshold D (Equation 7). We put all identified logs into the anomalous log set AL, which provides detailed clues to troubleshoot the system conveniently.

$$\text{center log} = \begin{cases} NormalLog, & abnScore \leq D \\ AnomalousLog, & abnScore > D. \end{cases} \tag{7}$$

V. Implementation Setup

A. Data collection

1) Infrastructure: Despite many log datasets being collected for research [3], [41], [42], [43], there is no open-source dataset documenting the evolution process. To fill this blank, we collect a new dataset LogEvol containing log data from the most widely-applied data processing system Spark [44] (LogEvol-Spark) and Hadoop [45] (LogEvol-Hadoop), across different versions.

To this end, we employ HiBench [46], a big data benchmark suite, to generate logs by running a set of workflows in Spark and Hadoop, respectively, from basic to sophisticated scenarios. In total, we run 22 workloads (shown in Table II) on the systems to cover more practical scenarios, while other existing datasets [43], [41] are collected from simply running two straightforward tasks (i.e., page rank and word count).

TABLE II
Workloads for collecting LogEvol.

Categories	Workloads
Micro task	Sort, Wordcount, etc.
Machine learning	Bayes Classification, Gradient Boosted Trees, etc.
SQL	Aggregation, Join, Scan etc.
Websearch	Pagerank
Graph	NWeight, Graph Pagerank
Streaming	Repartition

TABLE III
Statistics of LogEvol.

	Spark2	Spark3	Hadoop2	Hadoop3
# Logs	931,960	1,600,273	2,120,739	2,050,488
# Anomalous logs	1,702	2,430	35,072	30,309

Then, we repeat the procedure of running workloads using different versions of the software systems mentioned above, covering a wide time range and various data size scales. We select two typical versions of Spark (i.e., Spark2.4.0 and Spark3.0.3) and Hadoop (i.e., Hadoop2.10.2 and Hadoop3.3.3), as they have undergone systematic changes with significant differences.

2) Fault Injection: We inject 18 typical types of faults into the system to simulate real-world production failures: (1) *Process suspension*: Suspend processes in multiple types of nodes, one at a time; (2) *Process killing*: Kill processes in seven types of nodes, one at a time; (3) *Resource occupation*: Inject other computation programs to occupy CPU and memory; and (4) *Network faults*: Establish network faults such as losing packages, network delay, and connection lost.

In total, we collect 6,703,460 log messages (# Logs)with recognized 69,513 anomalous logs (# Anomalous logs), whose statistics are shown in Table III. To guarantee dataset quality, anomalous logs are discussed and annotated by two engineers who have two-year development experience with the Spark system. Since annotators have read a lot of logs in their development experience, they can provide reliable annotations.

B. Implementation details

In the multi-level representation extractor, we use BERT [23] as the pre-trained language model and fine-tune it with Hugging Face [47]. For the anomaly discriminator, we specifically choose leaky ReLU [48] as the activation function between two layers in the perceptron so as to resolve the "all-zero-solution" issue [49]. We set the dynamic threshold D to be 0.4 times of the maximum normality (hyper-sphere radius) in the training data in intra-version and 0.6 times for the inter-version. We set λ=0.5 in the experiments as the unitary and local features both serve as an important role in fault localization. We randomly split the collected logs into training, development, and testing sets for each software version with a standard 8:1:1 splitting. In contrast, the training set only contains logs collected in the fault-free periods as we assume the majority of logs are normal in a healthy system. All experiments are conducted in 64-bit CentOS 7 with Intel(R) Xeon(R) CPU and 1 GeForce RTX 2080 GPU for acceleration. It takes approximately 15 seconds for the anomaly discriminator to train in an epoch.

VI. Experiments

To evaluate the effectiveness of EvLog, we investigate three research questions:

RQ1: How effective is EvLog in identifying anomalous logs?
RQ2: How effective is EvLog in resolving evolving events and evolving sequences?
RQ3: How effective are different components in EvLog?

A. Experimental settings

1) Baselines: We select four unsupervised log-based analytics as baselines, including two anomalous log identification models and two anomaly detection (AD) models. LogAnomaly and LogSed are the state-of-the-art AD and log localization models, respectively. The reason why we choose AD baselines is, they both work for anomaly analysis with different granularities (i.e., coarse-grained and fine-grained); For AD models, we use the historical sequences to train a reference model and predict the next event as in the original papers. The actual next event that outside the predicted list of candidate events will be considered as anomalous due to its deviation from the reference model. In our implementation, we use the state-of-the-art log parser [19] to extract events for all baselines as they all require the parsing phase. In specific, we briefly characterize four baselines as follows.

- LOGAN [20] built the diagnosis system by constructing a directed graph from normal log event sequences. Then any of the test time logs that deviate from the directed graph will be considered anomalous.
- LogSed [6] addressed the interleaving logs problem by developing a two-stage approach to mine the important sequential relationship from log sequences. The incoming log message that violates that sequence will be regarded as anomalous.
- DeepLog [3] utilizes an LSTM network to capture sequential information of log data. It accepts the sequence of log event IDs to predict the next log, the actual log ID outside prediction will be regarded as an anomaly.
- LogAnomaly [36] is proposed for unsupervised anomaly detection with semantic representation for log events via an attention-based LSTM network.

2) Dataset: EvLog is evaluated on two datasets: a software evolution dataset collected from two representative systems (LogEvol) and a synthetic dataset (SynEvol).

LogEvol. Although existing study [12] analyzed the evolution process of Hadoop, and mentioned the importance of new-emerging log messages [38], there lacks a public dataset showing how logs change during software evolution. Hence, we evaluate our approach and compare it with baselines on the data collected in Section V-A. To our best knowledge, LogEvol is the first publicly accessible log dataset recording software evolution activities.

SynEvol. To evaluate how EvLog resolves the challenges of unseen events and unstable sequences separately (Note that EvLog is parser-free), we build a synthetic dataset based on the collected Spark2 logs in LogEvol (denoted as LogEvol-Spark2). Following previous work [38], we inject unseen events and unstable sequences into LogEvol-Spark2 to simulate the real-world software evolution as follows:

1. Unseen events are introduced by logging statement alteration in software updates. Developers may paraphrase or insert logging statements for customized functionalities. Since EvLog does not use a parser, we simulate the change by creating a set of synthetic log messages via (1) inserting, (2) deleting, or (3) replacing a common word from an original log message. Such modification is more likely to reflect the changes in log events.

2. Unstable sequences occur both in log generation and log evolution. Logs from multiple transaction flows may be interleaving, making the direct predecessor or successor of a certain log different. Moreover, log evolution is likely to cause variations via function ensemble or the changes of function invocation sequences. To construct synthetic sequences, we randomly remove a few unimportant log messages (far away from anomalous logs), repeat some log messages several times, or shuffle the log messages in a short time.

We inject the evolving events and unstable sequences into the original dataset, denoted as SynEvol-Events and SynEvol-Seqs correspondingly. The injection follows specific ratios. We inject the 5%, 10%, 15%, 25%, and 30% synthetic log messages and log sequences to LogEvol-Spark2, to observe how EvLog reacts to unseen and unstable sequences, respectively.

3) Evaluation metrics: To evaluate the effectiveness of EvLog in anomalous log identification, we apply Precision, Recall, and F1-score as evaluation metrics. In particular, Precision (P) is the percentage of logs that are correctly identified anomalous overall identified logs ($\frac{TP}{TP+FP}$). Recall (R) is the percentage of logs that are correctly identified anomalous over logs belonging to anomaly logs. ($\frac{TP}{TP+FN}$). F1 score (F1) is the harmonic mean of Precision and Recall ($2 * \frac{P*R}{P+R}$), where TP refers to the amount of anomalous logs that is correctly identified, FP refers to the number of normal logs that are wrongly predicted as anomalous, and FN means the number of anomalous logs that are identified as the normal logs.

B. RQ1: How effective is EvLog in identifying anomalous logs?

To evaluate how effective EvLog can pinpoint the anomalous logs with software evolution activities, we conduct experiments on our dataset LogEvol. The experiments engage two different settings: 1) Intra-version: identify the anomalous logs on the same system it is trained (e.g., Spark2 → Spark2); and 2) Inter-version: identify the anomalous logs in a different system version after training (e.g., Spark2 → Spark3).

We can draw two observations from the experimental results shown in Table IV. First, EvLog delivers an overall satisfactory performance under the intra-version setting with the average

TABLE IV
EXPERIMENTAL RESULTS IN IDENTIFYING ANOMALOUS LOGS (TRAIN SET→TEST SET).

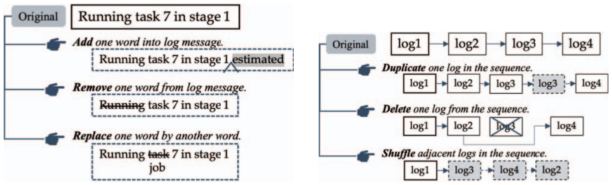
LOGEVOL-HADOOP												
	Intra-version						Inter-version					
	Hadoop2 → Hadoop2			Hadoop3 → Hadoop3			Hadoop2 → Hadoop3			Hadoop3 → Hadoop2		
Baseline	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.894	0.995	0.942	0.899	0.988	0.942	0.360	0.988	0.528	0.376	0.995	0.546
LogSed	0.910	0.995	0.951	0.925	0.986	0.955	0.371	0.988	0.540	0.390	0.993	0.560
DeepLog	0.913	0.985	0.947	0.926	1.000	0.961	0.386	0.999	0.556	0.410	0.971	0.576
LogAnomaly	0.926	0.994	0.958	0.939	0.988	0.963	0.389	0.998	0.560	0.407	0.995	0.578
EvLog	0.945	0.982	**0.963**	0.952	0.988	**0.970**	0.770	0.941	**0.847**	0.857	0.913	**0.884**

LOGEVOL-SPARK												
	Intra-version						Inter-version					
	Spark2 → Spark2			Spark3 → Spark3			Spark2 → Spark3			Spark3 → Spark2		
Baseline	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
LOGAN	0.798	0.943	0.865	0.967	0.870	0.916	0.016	0.943	0.032	0.012	0.943	0.024
LogSed	0.842	0.914	0.877	0.907	0.923	0.915	0.013	0.917	0.026	0.010	0.914	0.020
DeepLog	0.862	0.952	0.905	0.858	0.976	0.914	0.017	0.947	0.032	0.014	0.909	0.026
LogAnomaly	0.931	0.939	0.935	0.898	0.947	**0.922**	0.020	0.923	0.038	0.017	0.948	0.034
EvLog	0.970	0.974	**0.972**	0.944	0.888	0.915	0.922	0.700	**0.795**	0.920	0.812	**0.863**

F1 score of 0.967 in Hadoop and 0.944 in Spark, which is comparable with other baselines. The experimental results indicate that EvLog can learn the normality and effectively identify anomalous logs from log sequences. Besides, we find that deep learning-based approaches perform better than FSM-based approaches, demonstrating that neural networks are capable of capturing intrinsic sequential patterns and log semantics.

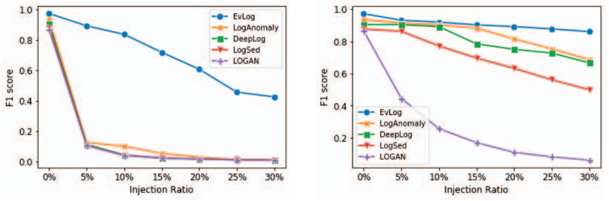
Second, in the inter-version scenario, EvLog significantly outperforms all baselines by a wide margin, demonstrating its effectiveness and robustness in software evolution. We observe that all baseline performances drastically drop (approximately an F1 score of 0.55) while EvLog achieves an average F1 score of 0.87 for Hadoop, which contains 3% new logs. In the case of Spark, where logging statement paraphrasing and insertion via software updating account for 10% logs, baseline performances are further significantly downgraded.

We analyze the reasons below. First, log parsers will generate unseen events when they encounter these new logs. Then, directed graph approaches (i.e., LOGAN, LogSed) and AD models (i.e., DeepLog, LogAnomaly) fail in matching these unseen events to any current events or predicted subsequent-event candidates. Consequently, current baselines label all unseen events as anomalous, leading to high false-positive rates (i.e., low precision). On the contrary, EvLog uses hierarchical clustering to learn abstract representations of log messages and aligns unseen events to similar past ones. In this way, the modified log message shares the consistent representation with its old one, so as to reduce false positives and improve anomalous log identification performance. Note that false positive rates in anomalous log identification, although not as severe as false negative cases, can still be problematic as they can lead to excessive work for maintainers.


Original
Running task 7 in stage 1
Add one word into log message.
Running task 7 in stage 1 estimated
Remove one word from log message.
~~Running~~ task 7 in stage 1
Replace one word by another word.
Running ~~task~~ 7 in stage 1
job
Original
log1
log2
log3
log4
Duplicate one log in the sequence.
log1
log2
log3
log3
log4
Delete one log from the sequence.
log1
log2
log3
log4
Shuffle adjacent logs in the sequence.
log1
log3
log4
log2

(a) SYNEVOL-Events example (b) SYNEVOL-Seqs example

Fig. 5. Examples of synthetic dataset SYNEVOL.



(a) Experiments in SYNEVOL-Events (b) Experiments in SYNEVOL-Seqs

Fig. 6. Experiment results on the synthetic dataset SYNEVOL.

Answers to RQ1: EvLog can effectively identify anomalous logs under both intra- and inter-version settings, all the while demonstrating its robustness and stability across software evolution activities.

C. RQ2: How effective is EvLog in resolving evolving events and evolving sequences?

We overcome the parsing errors challenge naturally since our model is parser-free. Thus, we are interested in how well our model addresses the other two challenges, i.e., evolving events and evolving sequences. To do so, we measure EvLog on the synthetic dataset, including SYNEVOL-Events and SYNEVOL-Seqs. Fig. 5 shows the examples in the dataset.

Fig. 6 shows the F1 scores of baselines, and ours under the injection ratio varies from 0% to 30% (the injection ratio of 30% means 30% of the original dataset was replaced by the synthetic one). The results demonstrate our approach's effectiveness in both evolving events and sequences compared with baselines. In particular, EvLog achieves the F1 scores of 0.42 and 0.86 in SynEvol-Events and SynEvol-Seqs, even though the synthetic dataset replaces 30% of the messages and sequences in LogEvol-Spark2, respectively. We attribute the advantage to the extracted multi-level semantics, as well as the stability of the normality learned by the anomaly discriminator.

Another observation is that log changes are more likely to damage the model's performance than sequence changes. This is because log changes bring unseen events to the trained model, posing greater difficulties for the model to deal with. On the one hand, our approach can still perform stably with evolving events due to EvLog's unique clustering mechanism that aligns old events with the new ones. This result is in line with our experiments in RQ1 that all baselines perform unsatisfactorily during version transferring, as many events are changed from Spark2 to Spark3. On the other hand, in terms of the unstable sequences, we conclude that neural networks (used by LogAnomaly, DeepLog, and ours), particularly those with the attention mechanism (used by LogAnomaly and ours), force the model to pay attention to the informative log messages while getting rid of unstable sequences.

Answers to RQ2: EvLog reveals the robustness across different types of changes happening in software evolution, owing to its multi-level semantics extractor and attention mechanism.

D. RQ3: How effective are different components in EvLog?

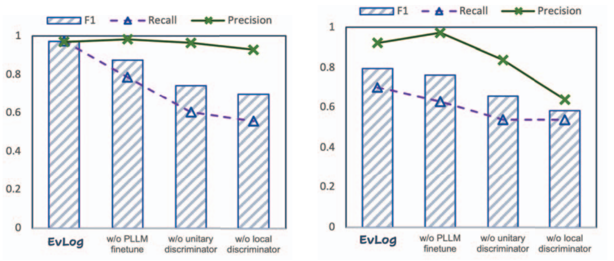
This research question investigates an ablation study on how much each design contributes to EvLog. Specifically, we remove each focused component one at a time and conduct experiments on LogEvol-Spark. In particular, we remove (1) the fine-tuning phase in PLM, (2) the unitary discriminator, and (3) the local discriminator, separately.

Our experiments in Fig. 7 show that all three components of EvLogcontribute to its effectiveness. The reasons based on the experiments are elaborated as follows. First, fine-tuning on the log dataset helps EvLogcapture precise semantics by bridging the knowledge gap between Spark domain knowledge and common sense knowledge. Second, the unitary discriminator, which operates on individual logs, learns the commonality of single normal logs. Third, removing the local discriminator largely degrades the overall performance since it provides a more comprehensive view of the contextual running status.

Answers to RQ3: The three components, i.e., PLM fine-tuning, unitary discriminator, and local discriminator, all show their effectiveness in the intended design of EvLog.

VII. Case study

This section conducts a case study (Fig. 8) to show how EvLog successfully deals with unseen events and avoids false positives. Having been trained on Spark2, baselines and EvLog are tested in the case from Spark3, where their AL predictions are marked with lights. Green, red lights refer to true positive and false positive, respectively. "GT" refers to the ground-truth AL set. All baselines wrongly predict the line2 and line3 logs as anomalous. We attribute the false-positive results on the two logs to their evolving events. In fact, this event is paraphrased as follows:



(a) Spark2 → Spark2

(b) Spark2 → Spark3

Fig. 7. Effectiveness of finetuning, unitary discriminator and local discriminator, respectively (train set→test set).

Spark2: Started reading broadcast variable <*>

Spark3: Started reading broadcast variable <*> with <*> pieces (estimated total size <*> MiB)

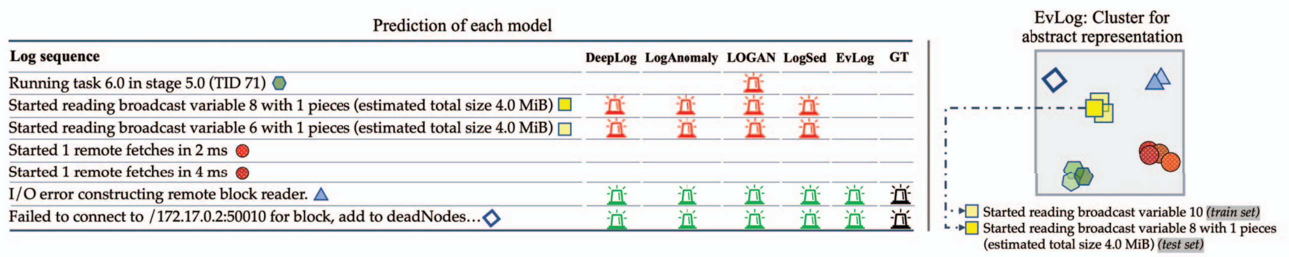
where <*> refers to the run-time generated numeric values.

Facing such evolving logs, EvLog can mitigate the associated issue by the abstract representation shown on the right-hand side of Figure 8. Though line2 and line3 are unseen logs, they can be assigned to a cluster that contains historical semantically similar log messages, according to their rich semantic representations. The yellow squares represent the rich representations of logs in the hyperspace, where the logs before and after paraphrasing stay closely in one cluster. Therefore, the high-level abstract representation remains stable in the change from the original logs to the paraphrased logs, and these new logs will not be mapped far away from the hyper-sphere's center. Eventually, the model can identify the new paraphrased log as a normal one because it does not deviate from the normality.

VIII. Threat to Validity

Internal threats. (1) Dynamic threshold. EvLog requires a dynamic threshold D to identify anomalous logs. Our study found that the satisfied threshold for intra-version and inter-version identification is 0.4 and 0.6 times the maximum normality in the training data, respectively. The threshold strikes a balance between recall rate and precision rate. In practice, maintainers can customize the threshold based on different scenarios. (2) Domain knowledge gap. Technical terms in logs may have specific meanings not captured by PLMs. For example, we use "volume" to describe a detachable block storage device in a computing system, but it usually refers to the degree of loudness or the amount of space in daily life. We fine-tune the PLM with the collected log messages to mitigate the threat.

External threats. (1) Software drastic evolution. Software systems possibly experience a drastic change, such as complete code restructuring or infrastructure renewal. In such scenarios, logging statements are likely to be altered significantly, and our approach has limitations to handle it without incremental



```
Running task 6.0 in stage 5.0 (TID 71)
Started reading broadcast variable 8 with 1 pieces (estimated total size 4.0 MiB)
Started reading broadcast variable 6 with 1 pieces (estimated total size 4.0 MiB)
Started 1 remote fetches in 2 ms
Started 1 remote fetches in 4 ms
I/O error constructing remote block reader.
Failed to connect to /172.17.0.2:50010 for block, add to deadNodes...
```

```
Started reading broadcast variable 10 (train set)
Started reading broadcast variable 8 with 1 pieces (estimated total size 4.0 MiB) (test set)
```

Fig. 8. An example of how EvLog identifies anomalous logs over software evolution.

learning. Nevertheless, our comparison between Spark2 and Spark3 over two years shows limited extreme changes. (2) Limited dataset. EvLog has been evaluated with only two real datasets and a synthetic dataset, and more real datasets with diverse job types are necessary to validate EvLog's effectiveness. However, as this is a brand-new task, datasets are sparse and challenging to collect. To address this issue, our created dataset is collected with representative 22 benchmark workloads from two widely-used systems. Although this dataset does not cover all possible workloads, it includes many commonly used ones and provides a practical simulation of the task.

IX. Related Work

A. Software evolution

Run-time data of systems can vary dramatically from time to time, as cloud systems are continuously being upgraded and evolving, causing variations in statistical properties [50]. Some studies [12], [38], [50], [51] noticed this issue and conducted empirical studies to investigate the effects it brings to automated techniques and found many methods are not intelligent enough to embrace such evolution. For example, previous research [12] found that the frequent source codes change liking releasing a new version leads to fragile log processing techniques. Also, LogTracker [51] revealed that it is challenging to request developers to maintain well-organized log statements as software evolves without rigorous specifications and demonstrated that the vast majority of context-similar logs come from log reversions.

These studies demonstrate that software evolution poses challenges to automated log analytics. Yet, they have not directly pointed out how and to what degree such an issue will affect log analytical tools. This paper is the first systematic study that fills such a gap by pointing out the three challenges log evolution brings and their reasons, as well as proposing a solution to overcome software evolution issues.

B. Failure analysis

Tremendous efforts have been devoted to cloud reliability insurance, and failure analysis has attracted considerable attention since it provides detailed clues for troubleshooting. Some existing approaches look deep into the source code to localize the failures, for example, mapping log messages to source code and reconstructing execution process for debugging [52]. However, source codes are not always accessible. Recently, log-based failure analysis is in the ascendant. To highlight the anomalous logs for failure diagnosis, existing approaches attempt to abstract the state transition processes in normal status by mining the logs and identifying the log that deviated from the model. For example, previous studies [20], [53] built a directed graph by regrading a log message sequence as an execution workflow and then checked whether logs in the test phase deviate from the graph. Besides, some studies use a retrieval-based approach to map a newly identified failure into the historical failure database whose cause is annotated by an expert in advance [54], [55].

However, these methods share three shortcomings in the evolution scenario. First, they rely strictly on dependencies within historical data. Second, they cannot extract sufficient semantics from logs, which are found significant in software evolution. Third, these approaches highly rely on prior expertise, making it impractical in modern evolving systems.

X. Conclusion

Existing advanced log localization models are proposed to discover anomalous logs that may indicate faults in a system automatically, but they ignore software evolution activities. This paper first empirically identifies three challenges (i.e., parser errors, evolving events, and unstable sequences) carried with software evolution and discusses how these challenges can affect localization models. Second, we propose EvLog to address the above challenges. To deal with the first two challenges, we develop a parser-free extractor to mine multi-level semantic representation from logs. Then, an anomaly discriminator with an attention mechanism is built to overcome the unstable sequence issue. At last, the effectiveness of EvLog in identifying anomalous logs over software evolution is confirmed by evaluating it on large-scale system logs. This is a newly identified research task in anomalous log localization due to software evolution, and the associated code of EvLog as well as the newly collected datasets, are released for research purposes. We hope our study can motivate more future work on software evolution in the log analytics community.

XI. Acknowledgement

The work described in this paper was supported by the National Natural Science Foundation of China (No. 62202511), the Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002), and the Key Program of Fundamental Research from Shenzhen Science and Technology Innovation Commission (No. JCYJ20200109113403826).

REFERENCES

[1] Shilpika, B. Lusch, M. Emani, V. Vishwanath, M. E. Papka, and K. Ma, "MELA: A visual analytics tool for studying multifidelity HPC system logs," in *3rd IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control, DAAC@SC, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 13–18. [Online]. Available: https://doi.org/10.1109/DAAC49578.2019.00008

[2] J. Liu, J. Huang, Y. Huo, Z. Jiang, J. Gu, Z. Chen, C. Feng, M. Yan, and M. R. Lyu, "Scalable and adaptive log-based anomaly detection with expert in the loop," *arXiv preprint arXiv:2306.05032*, 2023.

[3] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security, CCS, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[4] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller, "Data mining-based analysis of HPC center operations," in *Proceedings of the 19th IEEE International Conference on Cluster Computing, CLUSTER, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 766–773. [Online]. Available: https://doi.org/10.1109/CLUSTER.2017.23

[5] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin *et al.*, "Onion: identifying incident-indicating logs for cloud systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1253–1263.

[6] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *Proceedings of the 10th IEEE International Conference on Cloud Computing, CLOUD, Honolulu, HI, USA, June 25-30, 2017*. IEEE Computer Society, 2017, pp. 447–455. [Online]. Available: https://doi.org/10.1109/CLOUD.2017.64

[7] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*. IEEE Computer Society, 2009, pp. 588–597. [Online]. Available: https://doi.org/10.1109/ICDM.2009.19

[8] W. Meng, Y. Liu, S. Zhang, F. Zaiter, Y. Zhang, Y. Huang, Z. Yu, Y. Zhang, L. Song, M. Zhang *et al.*, "Logclass: Anomalous log identification and classification with partial labels," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1870–1884, 2021.

[9] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, "Using finite-state models for log differencing," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 49–59. [Online]. Available: https://doi.org/10.1145/3236024.3236069

[10] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Köprü, and T. Xie, "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 419–429. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678708

[11] M. M. Lehman and J. F. Ramil, "Software evolution and software evolution processes," *Ann. Softw. Eng.*, vol. 14, no. 1-4, pp. 275–309, 2002. [Online]. Available: https://doi.org/10.1023/A:1020557525901

[12] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. N. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *J. Softw. Evol. Process.*, vol. 26, no. 1, pp. 3–26, 2014. [Online]. Available: https://doi.org/10.1002/smr.1579

[13] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 102–112.

[14] B. Chen and Z. M. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, pp. 330–374, 2017.

[15] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services," in *Proceedings of the 24th IEEE International Conference on Web Services, ICWS, Honolulu, HI, USA, June 25-30, 2017*. IEEE, 2017, pp. 25–32. [Online]. Available: https://doi.org/10.1109/ICWS.2017.12

[16] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, "Digging deeper into cluster system logs for failure prediction and root cause diagnosis," in *Proceedings of the 16th IEEE International Conference on Cluster Computing, CLUSTER, Madrid, Spain, September 22-26, 2014*. IEEE, 2014, pp. 103–112. [Online]. Available: https://doi.org/10.1109/CLUSTER.2014.6968768

[17] X. Wang, X. Zhang, L. Li, S. He, H. Zhang, Y. Liu, L. Zheng, Y. Kang, Q. Lin, Y. Dang *et al.*, "Spine: a scalable log parser with feedback guidance," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1198–1208.

[18] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proceedings of the 24th IEEE International Conference on Web Services, ICWS, Honolulu, HI, USA, June 25-30, 2017*, I. Altintas and S. Chen, Eds. IEEE, 2017, pp. 33–40. [Online]. Available: https://doi.org/10.1109/ICWS.2017.13

[19] G. Chu, J. Wang, Q. Qi, H. Sun, S. Tao, and J. Liao, "Prefix-graph: A versatile log parsing approach merging prefix tree with probabilistic graph," in *Proceedings of the 37th International Conference on Data Engineering, ICDE, Virtual Event, 2021*. IEEE, 2021, pp. 2411–2422. [Online]. Available: https://ieeexplore.ieee.org/document/9458609

[20] B. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "LOGAN: problem diagnosis in the cloud using log-based reference models," in *Proceedings of the 4th IEEE International Conference on Cloud Engineering, IC2E, Berlin, Germany, April 4-8, 2016*. IEEE, 2016, pp. 62–67. [Online]. Available: https://doi.org/10.1109/IC2E.2016.12

[21] Y. Huo, Y. Su, C. Lee, and M. R. Lyu, "Semparser: A semantic parser for log analytics," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 881–893.

[22] Y. Huo, Y. Su, and M. Lyu, "Logvm: Variable semantics miner for log messages," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2022, pp. 124–125.

[23] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[24] V. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 492–504. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678773

[25] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 19th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 703–715. [Online]. Available: https://doi.org/10.1145/3468264.3468611

[26] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2279–2290.

[27] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2006–2018.

[28] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.

[29] L. Li, Z. Li, W. Zhang, J. Zhou, P. Wang, J. Wu, G. He, X. Zeng, Y. Deng, and T. Xie, "Clustering test steps in natural language toward automating test automation," in *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1285–1295. [Online]. Available: https://doi.org/10.1145/3368089.3417067

[30] M. Silic, G. Delac, and S. Srbljic, "Prediction of atomic web services reliability based on k-means clustering," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013, pp. 70–80. [Online]. Available: https://doi.org/10.1145/2491411.2491424

[31] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1448–1460.

[32] L. McInnes and J. Healy, "Accelerated hierarchical density based clustering," in *Proceedings of the 17th IEEE International Conference on Data Mining Workshops, ICDM Workshops, New Orleans, LA, USA, November 18-21, 2017*. IEEE Computer Society, 2017, pp. 33–42. [Online]. Available: https://doi.org/10.1109/ICDMW.2017.12

[33] P. Cifariello, P. Ferragina, and M. Ponza, "Wiser: A semantic approach for expert finding in academia based on entity linking," *Information Systems*, vol. 82, pp. 1–16, 2019.

[34] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[35] S. Na, L. Xumin, and G. Yong, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *2010 Third International Symposium on intelligent information technology and security informatics*. Ieee, 2010, pp. 63–67.

[36] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI, Macao, China, August 10-16, 2019*, S. Kraus, Ed. ijcai.org, 2019, pp. 4739–4745. [Online]. Available: https://doi.org/10.24963/ijcai.2019/658

[37] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[38] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, Long Beach, CA, USA, December 4-9, 2017*, 2017, pp. 5998–6008.

[41] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A large collection of system log datasets towards automated log analytics," *CoRR*, vol. abs/2008.06448, 2020. [Online]. Available: https://arxiv.org/abs/2008.06448

[42] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 25-28 June 2007, Edinburgh, UK, Proceedings*. IEEE Computer Society, 2007, pp. 575–584. [Online]. Available: https://doi.org/10.1109/DSN.2007.103

[43] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016, pp. 102–111. [Online]. Available: https://doi.org/10.1145/2889160.2889232

[44] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[45] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[46] "Hibench," Intel, 2021. [Online]. Available: https://github.com/Intel-bigdata/HiBench

[47] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[48] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[49] L. Ruff, N. Görnitz, L. Deecke, S. A. Siddiqui, R. A. Vandermeulen, A. Binder, E. Müller, and M. Kloft, "Deep one-class classification," in *Proceedings of the 35th International Conference on Machine Learning, ICML, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 4390–4399. [Online]. Available: http://proceedings.mlr.press/v80/ruff18a.html

[50] S. Kabinna, C. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 290–333, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9518-0

[51] S. Li, X. Niu, Z. Jia, J. Wang, H. He, and T. Wang, "Logtracker: learning log revision behaviors proactively from software evolution history," in *Proceedings of the 26th Conference on Program Comprehension, ICPC, Gothenburg, Sweden, May 27-28, 2018*. ACM, 2018, pp. 178–188. [Online]. Available: https://doi.org/10.1145/3196321.3196328

[52] A. R. Chen, "An empirical study on leveraging logs for debugging production failures," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 126–128. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00055

[53] A. Babenko, L. Mariani, and F. Pastore, "AVA: automated interpretation of dynamically detected anomalies," in *Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA, Chicago, IL, USA, July 19-23, 2009*. ACM, 2009, pp. 237–248. [Online]. Available: https://doi.org/10.1145/1572272.1572300

[54] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm?: automatic cause analysis for test alarms in system and integration testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 2017, pp. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2017.71

[55] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *Proceedings of the 41st International Conference on Software Engineering, ICSE, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 140–151. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00031