LogFlow: Simplified Log Analysis for Large Scale Systems

Marc Platini
University Grenoble Alpes, ATOS
marc.platini@atos.net

Thomas Ropars
University Grenoble Alpes
thomas.ropars@univ-grenoble-alpes.fr

Benoit Pelletier
ATOS
benoit.pelletier@atos.net

Noel De Palma
University Grenoble Alpes
noel.depalma@univ-grenoble-alpes.fr

ABSTRACT

Distributed infrastructures generate huge amount of logs that can provide useful information about the state of system, but that can be challenging to analyze. The paper presents LogFlow, a tool to help human operators in the analysis of logs by automatically constructing graphs of correlations between log entries. The core of LogFlow is an interpretable predictive model based on a Recurrent Neural Network augmented with a state-of-the-art attention layer from which correlations between log entries are deduced. To be able to deal with huge amount of data, LogFlow also relies on a new log parser algorithm that can be orders of magnitude faster than best existing log parsers. Experiments run with several system logs generated by Supercomputers and Cloud systems show that LogFlow is able to achieve more than 96% of accuracy in most cases.

1 INTRODUCTION

Large scale distributed systems, such as Cloud infrastructures and Supercomputers, are challenging to operate because the probability of experiencing anomalies and failures increases with the size and the complexity of the systems [11, 22]. The logs generated by these systems are the main source of information to understand the causes and consequences of a given event [11, 20, 24]. As such, several solutions, based on statistical and machine learning approaches [5, 7, 9, 10, 24, 26], or even based on deep learning [3, 7, 8, 18], have been proposed to analyze systems logs automatically. The goal of these works is failure detection [3, 7, 8, 18, 24], failure prediction [5, 10], or root cause analysis [9, 26].

We posit that, still today, the logs of large-scale systems are often analyzed *manually* by system administrators. Indeed, despite the fact that automatic log analysis is an active research area, it is also a young research domain where widely adopted solutions have not emerged yet [15]. Furthermore automatic log analysis can be complex because the information included in logs is not always accurate [6, 20]. A human expertise can be needed to correctly interpret the provided information or at least to label data before applying a learning approach [6, 8].

The goal of our work is to provide a tool that can help human operators in the daunting and tedious task of system logs analysis. To simplify and speed-up log analysis, we present LogFlow[1], a tool that automatically identifies correlations between log entries in the log journals of a large scale system. Given a selected log entry, LogFlow is able to construct a graph of correlations of the preceding log entries that can best explain the occurrence of the studied event. The work of Fu et al. [9] illustrates how such graphs can be of great help when running a root-cause analysis manually.

For LogFlow to be useful to practitioners, some properties are required. First, and most importantly, the obtained results should be reliable. It means that LogFlow should accurately identify correlations between log entries in system journals. Second, LogFlow should be fast. Human operators will be willing to use such a tool only if processing large volumes of new data can be done fast enough to avoid waiting for hours before getting results. This is a major challenge as large scale systems can produce huge amount of data each day. For instance, one of the supercomputers studied in this paper produces more that 2M log entries in 24 hours.

To accurately identify correlations between log entries, we leverage state-of-the-art deep learning techniques. More specifically, we use a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) [14] to build a model that is able to predict the next log entry based on a sequence of log entries. Furthermore, we augment it with an attention layer [16] to add interpretability to the model. We use an attention layer proposed in the context of Natural Language Processing [21] to understand which log entries the LSTM model relies on to predict the next log entry. Heuristics are used to deduce graphs of correlations between log entries from the outputs of the attention module.

Our model is not used to make *real* predictions. As the purpose of LogFlow is to help practitioners analyzing existing journals, when a prediction is run we already know the expected output. Hence, obtaining the correct prediction simply confirms that LogFlow has managed to identify strong correlations between log entries. On the other hand, as we explain in detail in the paper, we take advantage of the fact that we know the expected output of each prediction in the design of LogFlow. Namely, a difficult problem when building a predictive model for system logs is that the dataset is highly imbalanced: Some events may occur tens of millions of times while others occur only a few thousand times. If no specific measure is taken, there is a high chance that the model only learns to predict the frequently occurring events [4, 23]. In LogFlow, we propose a simple solution to this problem: Instead of having a single model in charge of predicting all log entries, we build several models, each of them being in charge of predicting a subset of the log entries. Then, knowing in advance the expected output allows us to select the appropriate model for each prediction. Our evaluation shows that, thanks to this solution, LogFlow achieves a very high prediction accuracy even for highly imbalanced datasets.

Regarding the performance challenge, we identify that the most expensive operation when trying to find correlations between log

[1] Available at https://github.com/bds-ailab/logflow

ICDCN'21, January 2021, Nara, Japan

entries is the step called *log parsing* [12, 17, 27]. Log parsing is the necessary first step for analyzing logs as it allows identifying all the log entries that report a *similar* event. Experiments run with the best existing log parsers show that they are able to parse a few thousands log entries per second. It implies that it can take hours or even days to parse very large journals.

LogFlow relies on a new parsing technique, LFP (for "LogFlow Parser"), that is based on the simple, yet conservative, assumption that all log entries representing the same event have the same number of tokens. This is in general realistic, as logs are generated based on message templates included in the code of applications, libraries, OS services, etc. [24]. This assumption, together with the use of a hashmap to sort data, allows achieving high performance by: i) creating an algorithm that has a linear time complexity; ii) enabling a simple parallelization of the program. LFP achieves orders of magnitude faster log parsing compared to state-of-the-art solutions [27], while being comparable in terms of accuracy.

To evaluate LogFlow, we use public datasets containing logs of Supercomputers and Cloud systems [13]. We also use our own dataset that corresponds to logs coming from the DKRZ (Deutsches Klimarechenzentrum) Supercomputer and that includes more than 600 million log entries. The results show that LogFlow is able to *predict* a given log entry in the journals with a precision and recall above 96% on all datasets, and thus, that we are able to accurately identify precursor log entries for a vast majority of the log entries. To verify the soundness of our approach, samples of graphs extracted by LogFlow have been validated by local experts. From performance point of view, our results show that LogFlow, thank to LFP, is able to parse any of the tested dataset in less than 3 minutes while best existing parsers can take up to several days.

To summarize, this paper proposes an original idea to help human operators in the analysis of logs of large scale systems. LogFlow reveals correlations between log entries in potentially very large log journals and relies on two main contributions to do so:

- To infer correlations between log entries, we demonstrate the efficiency of a deep-learning LSTM model augmented with a state-of-the-art attention layer. We also show how the fact that LogFlow is used to find correlations between past events can be used to build a more accurate learning model.
- To make our system usable even with very large journals, we propose a simple, yet accurate, log parsing strategy that can be orders of magnitude faster than existing log parsers.

We define the terminology used throughout this paper in Section 2, and discuss the related work. Section 3 presents and evaluates our solution based on deep learning to automatically find correlations between log entries in journals. Section 4 presents and evaluates our new log parsing technique. We draw conclusions from this work in Section 5.

2 BACKGROUND

2.1 Terminology

We use the word *journal* to name the file aggregating all system logs generated by one or multiple nodes of a large scale system. A journal is composed of *log entries*, ordered by their timestamp. A log entry is a message generated by one component of the system. A log entry is associated with a *template* which is defined by the tokens that remain the same in all *equivalent* log entries. Equivalent log entries correspond to the *same* system event reported by the same component and occurring at different times and/or in different locations in the system.

To illustrate the notion of template, let us consider the following two log entries reported by the Linux kernel:

```
CPU46: Package temperature above threshold
CPU17: Package temperature above threshold
```

These two log entries obviously have the same template:

```
*: Package temperature above threshold
```

and describe the *same* event, that is, the overheating of one CPU.

In this paper, we assume that a log entry is composed of a header that provides meta-information about the log entry such as its timestamp, the node and service that generated it, or its severity[2], and a free-text message which describes an event, as illustrated in the previous example. The parsing of a log entry refers to the parsing of its free-text message.

2.2 Related work

The analysis of the related work is divided into two parts. In a first part, we present existing work based on machine learning for log analysis in large scale systems to detect and predict anomalies. In a second step, we focus on the problem of log parsing.

2.2.1 Log analysis and fault prediction. Jauk et al. survey recent contributions to fault analysis and prediction in Supercomputers [15], and show that a significant number of publications focus on system logs. The automatic analysis of logs on Cloud environments is also an active research topic [7, 24].

Machine learning approaches have been studied as a means to analyze logs in large scale systems with the main goal of detecting or predicting failures [5, 24]. Logs are an important source of information to predict and understand failures. It should be noted, however, that the goal of LogFlow is not to automatically detect or predict the (future) abnormal state of the system, but to assist system administrators in the analysis of journals to discover the sequences of log entries that lead to a particular event. As a consequence, LogFlow is not designed to specifically predict rare events.

We are not the first to study the use of RNNs and, more specifically, LSTMs to analyze system logs [3, 6, 8]. LSTMs are a solution of choice to analyze logs as they are designed to model temporal sequences [14]. The work on Deeplog [8] was the first to propose such a solution to model the behavior of a large scale system using logs. Deeplog detects anomalies through a streaming analysis of logs entries. The high accuracy of the predictions made by DeepLog on Cloud journals illustrates the value of using a solution based on LSTMs in this context. The work by Das et al. [6] further demonstrates the ability of LSTMs to extract knowledge from system logs by: i) obtaining a high accuracy in the prediction of failures in supercomputers; ii) presenting a solution that predicts how much time is left before a failure occurs. Finally, in the context of cyber security, Brown et al. [3] propose to use an attention layer to better understand the LSTM model that is built by their system to detect attacks. Our work takes inspiration for this work and shows how a LSTM can be used together with an attention layer to deduce correlations between log entries.

[2] See, for instance, https://tools.ietf.org/html/rfc5424

As the core of their solution is about the construction of graphs of correlations between log entries, the work by Fu et al. [9] and Gainaru et al. [10] are close to LogFlow, although they are based on statistical approaches. As shown in previous works [6, 8] and as our evaluation confirms (see Section 3.3), solutions based on LSTMs can achieve a better accuracy than the techniques proposed in these work. Still, the work of Fu et al. [9] shows that identifying graphs of correlations between log entries can be of great help for system administrators in the difficult task of root cause analysis.

2.2.2 Log parsing. A typical usage scenario for LogFlow would be as follows. Anomalies are reported by the users of a large scale infrastructure and an administrator decides to analyze the journals generated over the past week to diagnose the health of the system and uses LogFlow to simplify this task. Here LogFlow can be useful only if it does not take hours to process these journals. Nence, to foster the use of the tool by system administrators, LogFlow should be able to process large volumes of data in a short period of time.

Using LSTMs to analyze system logs requires an initial pre-processing step called log parsing [27]. Based on the definitions given in Section 2.1, the purpose of a log parser is to identify the template associated with each log entry. This log-parsing step is the most time consuming for LogFlow when analyzing new log entries for a system.

The two main characteristics of log parsers are the accuracy, *i.e.*, their ability to discover all templates existing in journals, and their efficiency, *i.e.*, the speed at which they are able to parse journals. Zhu et al. present a extensive comparison of state-of-the-art log parsers [27]. Among the evaluated log parsers, Drain [12] obtained the best results in terms of accuracy, especially on the datasets that correspond to system logs. Furthermore, the results show that Drain is the most efficient open-source log parser for system logs, together with IPLoM [17]. Still, it should be noted that the journals we consider in this paper are much larger than the one used for performance evaluation in [27] (1GB vs hundreds of GBs). In the rest of the paper, we use Drain [12] as a reference for log parsing.

In Section 4, we present a new solution for log parsing, LFP, targeting system logs that is orders of magnitude more efficient than existing log parsers. Similarly to what was proposed in [17], the design of LFP is built on the assumption that all log entries corresponding to the same *event* include the same number of tokens. The log entries presented in Section 2.1 illustrate this assumption: In this case, it seems safe to assume that all log entries corresponding to a CPU overheating event reported by the Linux kernel include 5 tokens. Our algorithm makes use of this assumption to allow the parallel processing of large journals. It also leverages a hashmap data structure to achieve a linear practical time complexity.

3 FINDING CORRELATIONS BETWEEN LOG ENTRIES

This section presents the core of LogFlow, our tool to automatically identify correlations between log entries in system journals. LogFlow relies on a model to predict the next log entry in systems logs based on the sequence of the most recent log entries. This model is built using an LSTM [14] and is augmented with an attention layer [21] to obtain information about the log entries that most contributed to the prediction. From the outputs of the attention layer, LogFlow deduces correlations between log entries.

We start by presenting the design of LogFlow and how it discovers correlations between log entries in journals. Then, using logs from real systems, we present an evaluation of the predictive capabilities of LogFlow, and so, of its ability to find valid correlations.

3.1 Building the model and detecting correlations

This section describes how we build a model based on an LSTM to predict log entries in system journals and how we deduce correlations between log entries from the output of the model. We start by presenting the necessary pre-processing steps that should be applied to the data before being given as input to the LSTM. Then, we detail the architecture of our predictive model. We present how we take advantage from the fact that LogFlow already knows the expected output when it makes a prediction to deal with the problem of *imbalanced dataset*. Finally, we describe how we process the outputs of the attention layer integrated in our predictive model to pinpoint important correlations between log entries.

3.1.1 Pre-processing log entries . Our solution relies on an LSTM for predictions. However, the LSTM cannot directly use raw log entries as input [8]. Instead, similarly to what has been done in preceding works [6, 8], pre-processing steps involving log parsing and embedding are applied:

Log parsing. As already described, log parsing aims at discovering templates in the log entries to identify all log entries that describe *equivalent* events. Accurately identifying templates is the first necessary step to build a model that can learn sequences of events occurring in the system. In the following, we use the state-of-the-art log parser called Drain [12] for this step. We propose an alternative log parsing strategy in Section 4.

Embedding. The purpose of the embedding step is to replace templates identified by the log parser by numerical vectors that can be easily processed by a deep neural network. To this end, LogFlow relies on the state-of-the-art algorithm called `word2vec` [19]. When dealing with words in the context of Natural Language Processing, `word2vec` builds vectors that carry semantic meaning. Hence, these vectors allow a machine learning algorithm to infer that although *computer* and *server* are different words, they might have a similar meaning. In our case, the inputs of `word2vec` are templates instead of words. This way, `word2vec` takes into account the sequences of log entries in which each template appear to try identifying *similarities* between templates.

3.1.2 The predictive model. Considering the large number of log entries in the journals of large scale infrastructures, we choose to use a neural network, more precisely an LSTM [14], to build a model that can predict the next log entry from a sequence of log entries. LSTMs are neural networks dedicated to learn temporal correlations. They have been shown to be highly accurate for Natural Language Processing [2]. LSTMs model the natural language as sequences of words. From this point of view, analyzing sequences of log entries in system journals can be seen as a similar problem.

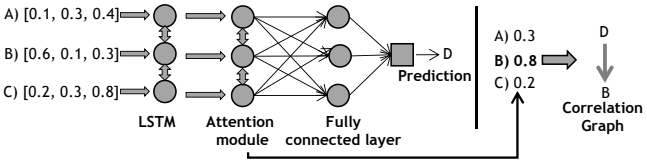


Figure 1: Model architecture and identification of correlations between log entries

For this model to be useful for LogFlow, it should be able to accurately predict events but it should also be interpretable, that is, it should provide information about the entries the model focuses on to make predictions. To this end, we use attention module. This module associates a weight with each input. A higher weight means that the input is more important in the prediction. Different solutions have been proposed for getting attention with LSTMs [16, 21]. We choose to use the Temporal Attention Gated Model [21] because of its good results on multiple benchmarks and its simple design. The attention module is trained as a part of the model.

The last layer of our architecture is a fully connected layer that aggregates the output of the attention module to make predictions. More precisely, the output of the fully connected layer is, for each identified template, the probability estimated by the model that this template is the one that will appear after the sequence of log entries provided as input. We consider that the predicted template is the one with the highest probability.

Figure 1 summarizes the model architecture and presents through an example how correlations between log entries are deduced from the outputs of the attention layer after a correct prediction. In this example, we assume that a journal includes a sequence of 4 log entries that correspond to different templates that we name A, B, C, and D. To understand whether log entry D has correlations with the 3 preceding log entries, we ask the trained model to predict the next event when it is given the sequence A–B–C as input. As illustrated on the figure, what is given as input to the LSTM is actually neither the log entries directly nor the templates identified by the log parser, but the embedding vectors representing the templates obtained after the pre-processing steps. The model correctly predicts D. Hence, we conclude that there are correlations between D, and the 3 preceding log entries. To obtain more information about these correlations, we analyze the weights provided by the attention module. The high weight associated with B reveals that the model mostly relied on this input to predict D. Hence, we deduce that there is a strong correlation between log entries D and B.

3.1.3 Dealing with imbalanced datasets . The description given until now assumes that a single model is in charge of all the predictions. However, as shown in Section 3.3, some templates may appear way more often than others in system logs. For instance, one can easily anticipate that log entries reporting network connections will (hopefully) appear more often in journals than events reporting that a hard disk is full. Having such an imbalanced dataset is an important issue in machine learning [4, 23]. The risk is that the model tends to focus on correctly predicting the most common events to quickly decrease its error rate but that it consequently becomes unable to predict less common events. The problem has already been identified for predictions with system logs in [10].

Diverse solutions have been proposed in the literature to deal with the problem of class imbalance with deep neural networks [4]. They range from adapted sampling methods to the definition of new loss functions. Such solutions can achieve good results but require a high degree of expertise [4].

To deal with this problem, we propose an alternative method that takes advantage of the fact that, for the predictions made by LogFlow, we know in advance the expected output as we illustrated in Figure 1. Our solution is inspired from the *Single-class learning* approach [23]. This method deals with imbalance for binary classification problems by building a model only for the class with the lowest number of occurrences. In our case, we propose to build several models, each dedicated to a subset of the templates/events. Since the total number of events can be large, we cannot afford to build one model per event. We build several models, each being in charge of the predictions for a set of templates that appear a similar number of times in the journals. More specifically, if an event appears in the order of 10^c times, we say that it has a cardinality C. We build a single model for all events that have the same cardinality.

To fully describe our solution, we should mention that during the training phase, the model built for events with cardinality C_a is given as input only sequences of events $[e_o, e_1, \ldots, e_n]$ leading to an event e_{n+1} with cardinality C_a. Of course, the events e_i in the provided sequences do not necessarily have a cardinality C_a.

Using such an approach is only possible in the specific context of LogFlow. In general, when running a prediction, the expected output is not known beforehand. Hence it is not be possible to know which model to use for a given prediction. Note also that our solution based on multiple models allows us to easily increase the parallelism during the training phase. This is important as it is known to be challenging to find parallelism in the training of LSTMs models [1]. The results presented in Section 3.3 show that the proposed approach greatly improves the prediction accuracy on events that appear less often in the journals.

3.1.4 Extracting correlations from the outputs of the attention layer and building graphs of correlations. As shown in Figure 1, LogFlow identifies correlations based on the outputs of the attention module. This module gives the weight of each previous log entry in a prediction. From these weights LogFlow needs to decide what is a significant enough weight to conclude about a correlation. Furthermore to build the graph of correlations between a selected log entry l, the algorithm is executed multiple times. Namely, the algorithm is executed a first time considering entry l and gives as output a set of log entries L_{cor} that can best explain l. Then, to extend the graph, the algorithm is executed for each log entry l' in L_{cor} to find the log entries that can best explain the log entries from L_{cor}. Conditions to decide when it becomes useless to dig deeper in the graph of correlations need to be defined. We define automatic rules to take decisions about the construction of the graphs, and thus, avoid asking users to set thresholds for these two problems.

To decide when a weight provided by the attention module is significant enough, we automatically compute a threshold T_{weight}. This threshold is computed as: $T_{weight} = mean(W) + stddev(W)$, where W is the set of weights provided by the attention module for a given prediction. Using this threshold, we select only the log entries with a weight greater than the sum of average and standard

deviation of all of weights returned by the attention module for a prediction, that is, we select only the remarkable weight values. The value of this threshold is recomputed for each prediction.

Regarding the construction of the graph for a selected log entry, we consider that the graph is complete when no new nodes are added while trying to find new correlations. The first reason to stop adding nodes is when LogFlow makes a wrong prediction. A wrong prediction means that the model could not capture correlations between the currently analyzed log entry and the previous log entries in the journal. The second reason is when the templates associated with the log entries that are identified as having correlations with the currently analyzed log entry, are already represented in the graph. In the case, we simply add an edge to represent the new correlation without creating a new node. Note that this second condition can be deactivated for users that would like to try digging deeper in the graph of correlations to keep as only stopping condition the case where wrong predictions are made.

3.2 Using LogFlow to analyze new log entries

Using LogFlow to analyze the journals of a large scale infrastructure involves first a learning step, during which the predictive model is built using the available system journals of the studied infrastructure. Once the model is built, it can be used to identify the correlations between the log entries in any part of the journals.

To analyze new journals, that is, log entries generated after the model has been built, no training phase is required. By default, the only steps to execute before being able to use the model are the pre-processing steps. Log parsing is required to identify which template each log entry corresponds to. Hence, the delay required by LogFlow to provide information about correlations for new log entries only depends on the efficiency of the log parser.

This workflow assumes that all the templates appearing in the new log entries are already present in the log entries used for training the model. This should be the case if the journals used for training cover a period that is large enough and if there is no system update. If new templates do appear, it is then required to go first through a new training phase to update the predictive model.

3.3 Evaluation

The evaluation of LogFlow focuses on assessing its prediction capabilities. Indeed, if the model is able to predict the next log entry based on the previous ones, then we can be confident that it found valid correlations. Four datasets representing the journals of real systems are used for the evaluation.

3.3.1 Methodology. This section presents the four datasets and the metrics used for our evaluation. It also describes the hardware on which the experiments are run and the software configuration.

Metrics. Since log prediction is a multi-class classification problem where each event corresponds to a class, grading the predictive capabilities of our model requires adapted metrics. The metrics used for binary problems are the recall[3] $R = TP/(TP+FN)$ and the precision $P = TP/(TP+FP)$, the F_1 score being a measure of the accuracy that combines recall and precision $F_1 = 2\times(P\times R)/(P+R)$. Instead of using the standard version of these metrics, we choose to measure *micro*-metrics and *macro*-metrics [25], that can better capture the prediction accuracy for multi-class problems. Namely, precision and recalls are computed for each class individually, and micro-metrics are obtained by computing a weighted average on the results while macro-metrics are obtained by computing an unweighted average. Hence micro-metrics evaluate the ability of making a maximum of correct predictions while macro-metrics evaluate the ability to correctly predict all classes.

Datasets. The Loghub repository [13] is a great source of data regarding system logs. Among the datasets referenced by Loghub, we select 3 main datasets, `BlueGene/L` (BGL), `Thunderbird` (THU), and `Spark` (SK) based on the following criteria: i) They are logs of large scale systems (Supercomputers and Clouds); ii) They are of significant size (700MB for BGL, 2.75GB for Spark, 29.6GB for Thunderbird); iii) They include a large number of templates (>100).

Additionally, we use a dataset corresponding to 10 months of operation (from January to September 2018) of the DKRZ Supercomputer[4]. This dataset includes 72GB of data and more than 1400 templates. It is representative of today's large scale systems. Unfortunately, this dataset could not be made publicly available.

Modifications of the datasets. To accurately evaluate the predictive capabilities of LogFlow, we apply some filtering steps on the datasets.

In the datasets, we observe sequences including only (or mostly) log entries with the same template. A high accuracy in the predictions for such sequences can easily be obtained by predicting that the next log entry is the same as the previous one. To avoid obtaining artificially high accuracy because of these cases, we apply a filtering step. Namely, when predicting an event `E`, we provide as input a window where log entries corresponding also to template `E` have been filtered out[5]. This filtering step is applied both when training the models, and when running predictions.

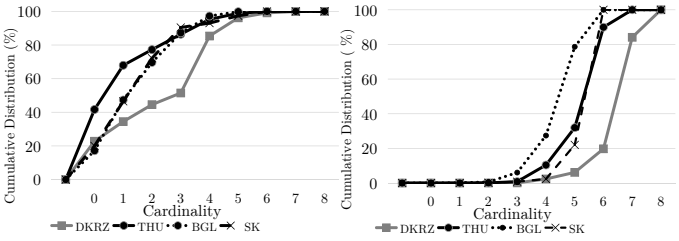
Our second filtering step removes the log entries that appear too often in the journals. Using the definition of cardinality introduced in Section 3.1.3, Figure 2 presents the distribution of the templates and the log entries according to their cardinality. Some templates appear millions of times while others only appear a few times. We filter out the log entries associated with a template that appears more than 10^7 times in a dataset for three main reasons: i) Few templates belong to this category as shown in Figure 2a (*e.g.*, only 2 in the Thunderbird dataset), which implies that with our multi-model approach, the models in charge of these cardinality could manage to make correct predictions without evidencing strong correlations; ii) These events appear so frequently that they mostly do not carry any useful information for the user and that they have very low correlations with other log entries; iii) Accurately predicting these events would dramatically improve the statistics about accuracy and hide the real performance of our models.

Finally, we do not try to predict events that appear less than 10^3 times in the logs. We make this choice because deep-learning methods (including LSTMs) require a large number of examples to provide accurate results. Predicting correctly such log entries would most probably be a matter of luck or overfitting, and would not

[3] TP: True positive; FN: False negative; FP: False positive.

[4] Ranked 93 in the Top500 of June 2020.

[5] However, if an event E' appears multiple time in the window, the multiple log entries corresponding to this template are kept.



(a) Templates (b) Log entries

Figure 2: Empirical Cumulative Distribution Function of the number of templates and the number of log entries according to the cardinality.

give any reliable information about correlations. We further discuss how to handle this kind of events in Section 3.5. One may think that removing these events makes a comparison with the related work unfair. However Figure 2b shows that these templates represent a low number of log entries. Hence including these events in our evaluation could modify the micro-metrics (main metric considered by the related work) by at most 0.62% based on our computations.

During our experiments, we use 60% of the data for training and 40% for testing. To generate the testing and the learning set, we shuffle the whole log entries in a dataset and we split them into two sets. Obviously, once the log entries to be used for learning and testing are selected, the non-shuffled data are used to provide the sequences of log entries as input to the model.

Hardware and software configuration. LogFlow is implemented using Python3[6]. We use the `word2vec` implementation provided by Google[7]. Pytorch is used to develop the LSTM and the attention module. Regarding the log parser, we use the implementation of Drain provided by LogPAI[8]. The LSTM is trained using a node with 2 20-core Intel Xeon SKL Gold 6148 CPU, 384 GB of RAM, and one Nvidia V100 GPU (CUDA v.10.1, Nvidia Driver v.418.67). The operating system is RedHat v.7.6.

The following parameters are used for the model. We use a window of 30 previous log entries to train the model. The impact of this choice is evaluated in a dedicated experiment. We use a batch size of 128. The learning model is composed of one unidirectional LSTM with a hidden size of 50 and 1 layer followed by the attention layer and one fully connected layer of size 50. We set the length of the vector provided by the `word2vec` algorithm to 20. Most of these parameters could be optimized to try obtaining better predictions. However, as our results show, this configuration already achieves good results on all datasets. The condition to stop the learning process is when the *macro-*F_1 value does not increase by more than 0.01 compared to the 3 previous iterations. Using this stopping condition, training the models takes approximately 2 hours.

3.3.2 *Evaluation of the predictive model.*

Accuracy of the predictions. Figure 3 presents the micro-metrics and macro-metrics for the predictions made on all the datasets.

[6] https://github.com/bds-ailab/logflow
[7] https://code.google.com/archive/p/word2vec/
[8] https://github.com/logpai/logparsercommit62c7600

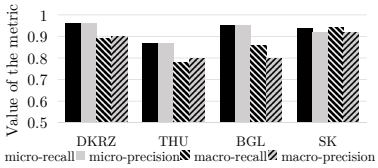
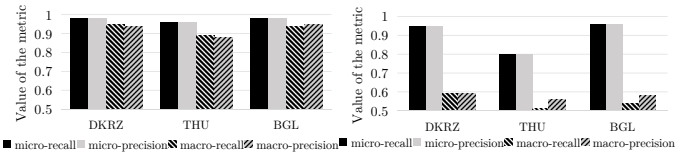


Figure 3: Predictions accuracy of LogFlow



(a) LogFlow (b) Using a single model

Figure 4: Prediction accuracy with a journal per node

To improve the readability, the y-axis starts at 50%. The results demonstrate the high prediction capabilities of our model on all datasets. Considering the micro-metrics, we observe that in the worth case, 86% of precision and recall are achieved, and that the results are as high as 96% on the DKRZ dataset.

On BGL, our micro-precision and micro-recall reach 95%. This dataset was also used by the close related works of Fu et al. [9] and Gainaru et al. [10]. On this dataset, Gainaru et al. obtain a micro-precision of ~90% but a micro-recall of only ~72%. Fu et al. achieve a micro-precision of ~88% and a micro-recall of ~75%. This shows the advantages of using a LSTM.

Figure 3 also shows that, in general, the value of the macro-metrics are lower than the ones of the micro-metrics. This means some events that occur less often are not predicted as accurately. We further analyze this problem later in the section.

Impact of aggregating journals from multiple nodes. The results presented until now consider the case where the log entries generated by all nodes in the system are aggregated into a single journal. This can make the prediction task difficult as unrelated log entries generated by different nodes might be interleaved.

To measure the impact of this problem, we run an evaluation where the log entries are sorted to create one journal per node, and where the prediction are made based on these per-node journals. The `Spark` logs are not considered for this evaluation because they do not include information about the node that generated a log entry. The results presented in Figure 4a show that the predictions are much better in this case. The micro-metrics reach at least 96% on all datasets while the value of the *macro*-metrics is also greater than 0.94 on the BGL and the DKRZ datasets. Hence, creating a separated journal per node allows achieving much better predictions on average, even if it might prevent from detecting correlations between events occurring on different nodes.

Multi-model evaluation. To evaluate the impact of our choice to build multiple predictive models (one per set of events that have the *same* cardinality), we evaluate a solution that uses a single model. The results presented in Figure 4b, re-using the per-node journals created previously, show that the *micro*-metrics values

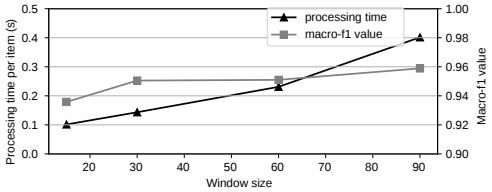


Figure 5: Impact of the window size on the *macro*-F_1 value and on the processing time.

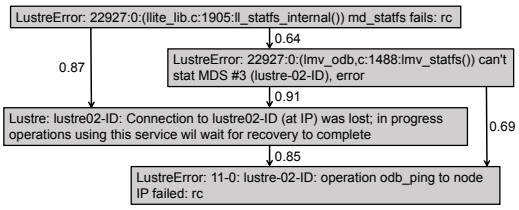


Figure 6: Example of graph obtained on the DKRZ dataset

remain good but *macro*-metrics values are significantly impacted (reduction by more than 30%). This means that a single model is not able to correctly predict templates that appear less often in the logs, and demonstrates the benefits of using multiple models.

Detailed analysis. The results presented until now show that the value of the macro-metrics are lower that the value of the micro-metrics. To better assess the consequences of this observation, we conducted a more detailed analysis. We present this analysis with the DKRZ dataset because for this dataset we can rely on the knowledge of our in-house experts. Note however that the results obtained with the other datasets are qualitatively the same.

We studied the predictions according to the templates appearing in the DKRZ journals. For a large majority of the templates (88%), a precision and recall higher than 90% is achieved. On the other hand, a precision lower than 60% is observed for only 5.1% of the templates and a recall lower than 60% for only 4.2% of them. In both cases, these templates correspond to less than 0.7% of the log entries. An example of templates for which the prediction results are very low is "`pps_core: LinuxPPS API ver. 1 registered`". The template is not well predicted probably because it corresponds to the first log entry written when this software starts.

We also analyzed the value of the macro-metrics according to the service that generated the log entries and according to the severity of the log entries. We observed that good recalls and precision were obtained for all levels of severity. On the other hand, three services (`slurmstepd`, `sssd`, and `crond`) have macro-metrics between 50% and 80%. It can be noted that 2 of these services relates to external actions, while the last one relates to automatic actions.

3.3.3 Impact of the window size. Among the parameters of our models, one that could have a significant impact on the correlations that are extracted is the window size, that is, the number of preceding log entries that are provided as input to make a prediction.

Figure 5 assesses the impact of the window size on the training time and on the macro-F_1 value. We present the results for the DKRZ dataset, but the results are qualitatively the same on all the datasets. The macro-F_1 value increases significantly when increasing the size of the window from 15 to 30 but increases by less than 0.01 after that. On the other hand, the processing time increases significantly when increasing the window size (here 4 models are trained in parallel on the GPU). Hence, the default size of 30 provides a good trade-off between accuracy and training time.

3.3.4 Inference time. We evaluated the time required by LogFlow to identify the log entries correlated with a selected log entry, once the model has been trained. Measurements run on a laptop equipped with a 4-core i7-7820HQ CPU and 32GB of memory show that running a prediction and analyzing the outputs of the attention layer for one log entry takes less than 10ms, which is compatible with an interactive use of LogFlow.

3.4 Example of LogFlow graph

Figure 6 presents a graph obtained using LogFlow on the DKRZ dataset, that has been validated by local Lustre experts. Here, IP addresses and host names have been anonymized. The nodes of the graph are the log entries and the edges show the correlations between the log entries. The numbers on the edges correspond to the weights provided by the attention layer. The higher the weight, the stronger the correlations between the log entries.

Starting from the log entry "`md_statfs fails: rc`", LogFlow tries to *predict* it from the previous log entries. The prediction is correct and LogFlow identifies two main correlated logs: "`can't stat MDS #3`" and "`Connect to lustre02-ID was lost`". LogFlow then tries to predict these two log entries to discover more correlations with previous log entries. This graph shows that the error "`md_statfs fails: rc`" might be related to a network issue ("`obd_ping to node IP failed`").

3.5 Discussion

The presented results demonstrate the capacity of LogFlow to predict the next log entry based on the current sequence of log entries, and so, the ability of the obtained model to find correlations between log entries. Currently LogFlow ignores very rare events because neural networks need to have enough samples of one class to build an accurate model. To understand if this can be a strong limitation, we analyzed the log entries corresponding to rare events in the DKRZ dataset manually. A majority is generated when a node starts. During this phase, sequences of logs are easy to read without the help of a tool such as LogFlow. To deal with the remaining rare events (rare failure events for example), it would be interesting to study how algorithms proposed by related works [9, 10] could be combined with LogFlow to detect correlations for these cases.

Our evaluation reveals that the current version of LogFlow provides better results when log entries are aggregated per node. Even if the results on the merged logs are better than the one achieved by related works, it shows that LogFlow could be improved to better identify correlations between events occurring on different nodes of the system. Some related works [5, 9] propose to use information about the jobs that generated the log entries or the physical location of the nodes in the infrastructure to better analyze the correlations between the log entries generated on different nodes. Integrating such strategies is part of our future work.

4 EFFICIENT LOG PARSING

The section presents a new log parsing algorithm. The evaluation presented in Section 3.3 are run using the log parser Drain [12]. According to extensive evaluations [27], Drain not only is the most accurate log parser in the state of the art, it is also among the most efficient ones. Efficiency is a major concern because the systems we consider generate huge volumes of data. For instance, over the period studied in this paper, the DKRZ supercomputer generates 2M log entries per day. Our evaluation shows that Drain is not able to cope with such high numbers of log entries (see Section 4.2).

4.1 The design of LFP

The purpose of a log parser is to identify *variable* and *constant* parts in log entries to derive *templates*. If we have two log entries `"Connection from NodeA"` and `"Connection from NodeB"`, a log parser should be able to identify that they correspond to the same template `"Connection from *"`. Hence, log parsing can be seen as a clustering problem where all log entries associated with the same template should be put in the same group [12].

To achieve high performance in log parsing, the design of LFP aims at achieving the following goals: i) Having a simple representation of the tokens appearing in log entries to reduce the memory cost associated with storing these tokens; ii) Limiting the time complexity of the algorithm; iii) Enabling parallelism.

4.1.1 Efficient representation of tokens. The first step of LFP is creating a compact representation of tokens that still allows meaningful comparisons. This step is also introduced in some other parsers [12]. By *meaningful*, we refer to the fact that using a strict equality for comparing tokens would be too restrictive. If we take two memory addresses, `0x0c35685d` and `0x200da20c`, with a strict equality, we would simply consider them as different tokens, and so, we may conclude that the token is simply a *variable* part in a log template. But we think that a more accurate template should consider this token is a memory address.

Drain uses regular expressions to deal with this problem [12]. To avoid depending on the definition of regular expressions by the user, we propose an alternative representation with the goal of being able to simply identify equivalent tokens that correspond to objects that commonly appear in computer logs (memory addresses, file paths, IP addresses, etc.). Hence, we create token descriptors using three rules: i) For tokens including only letters, the descriptor is the token itself; ii) For tokens including only numerical characters, the descriptor is the constant `NB`; iii) For all other tokens, the descriptor is a vector including 5 entries. The first 4 entries are boolean values describing the presence of a type of character: numerical characters, uppercase letters, lowercase letters, non alpha-numeric characters. The last entry is the length of the word. Obviously, the descriptors of tokens consume less memory than the tokens themselves, especially for complex tokens such as file paths.

We should also mention that to identify tokens in log entries, the white space is used as default separator, but we also considers the characters ":" and "=" as separators. To illustrate the transformations applied during this first stage of the log parser, we can consider the following set of log entries:

```
1) Temperature_Celsius changed from 55 to 54
2) Connection of user=R52 from Moon
3) Temperature_Celsius changed from 54 to 53
4) Connection of user=B782 from Moon
5) Connection of user=Felix from Mars
```

After this stage, the log entries include the following descriptors:

```
1) (0,1,1,1,19) changed from NB to NB
2) Connection of user (1,0,1,0,3) from Moon
3) (0,1,1,1,19) changed from NB to NB
4) Connection of user (1,0,1,0,4) from Moon
5) Connection of user Felix from Mars
```

4.1.2 Identifying templates using comparison vectors. To identify the log entries that correspond to the same template, we propose to rely on *comparison vectors*. For a given log entry e of size n including the descriptors $d_{e1}, d_{e2} \dots d_{en}$, its comparison vector V_e is defined as follows: $V_e = [v_{e1}, v_{e2}, \dots v_{en}]$, where v_{ex} is the number of times the descriptor d_{ex} also appears in position x in other log entries.

Once the comparison vector of a log entry has been computed, its template can directly be deduced from this vector. The rule for identifying the set of *constant* descriptors $D_{template}$ describing the template associated with log entry e based on vector $V_e = [v_{e1}, v_{e2}, \dots v_{en}]$ is as follows. Let v_{most} be the value that appears the most frequently among $v_{e1}, v_{e2}, \dots v_{en}$. A descriptor d_{ex} is in $D_{template}$ if and only if $v_{ex} = v_{most}$.

Applying this algorithm to the set of log entries introduced above, we obtain the following comparison vectors:

```
1) [1, 1, 1, 1, 1, 1]
2) [2, 2, 2, 0, 2, 1]
3) [1, 1, 1, 1, 1, 1]
4) [2, 2, 2, 0, 2, 1]
5) [2, 2, 2, 0, 2, 0]
```

The comparison vectors of entries 1, 3 only include the value 1. Hence, their template include all the descriptors associated with these entries. For log entries 2, 4, and 5, the value that appears the most often in the comparison vector is 2. Hence the constant part of their template include 4 descriptors (corresponding to the entries with value 2), and they all correspond to the template `"Connection of user * from *"`, where "$*$" represents the *variable* parts.

4.1.3 Efficient implementation of the algorithm. To obtain an efficient implementation, we rely first on the assumption that log entries including different number of tokens correspond to different templates. Note that this assumption is also exploited in other solutions [12, 17], but is used only to reduce the complexity of the algorithm and not to introduce parallelism. Hence, processing a set of log entries starts by grouping the log entries according to their number of tokens. Then each group can be processed independently and fully in parallel. We should point out that the simple identification of templates used in our algorithm, based on comparison vectors that checks whether a token appears at the exact same position in different log entries, also relies on this assumption.

For a group including N log entries with size S tokens, the definition of comparison vectors implies that the total number of comparisons required to build all comparison vectors is $N^2 \times S$. To reduce this number in practice, we rely on a hashmap data structure. The keys in the hashmap are pairs composed of a descriptor and a position of the descriptor in the log entry, while the value associated with a key is the number of time this descriptor appears at the given position in the log entries.

Hence, the algorithm processes the log entries in a group sequentially to replace the tokens by the corresponding descriptor,

and when a descriptor d is created at position p in a log entry, the following algorithm is executed:

```
key = hash(d + "-" + p)
if key is in hashmap:
    hashmap[key] += 1
else:
    hashmap[key] = 0  // create an entry for the new key
```

When all log entries have been parsed and the hashmap is fully populated, constructing the comparison vector of one log entry only requires S accesses to the hashmap. Hence, building the comparison vectors for all log entries in the group only requires $N \times S$ executions of the hash function and memory accesses. The same cost is associated with populating the hashmap.

4.2 Evaluation

To evaluate the log parser, we use the same datasets as in Section 3.3. Our experiments evaluate the efficiency and the accuracy of LFP. In a second step, we evaluate the impact of the log parser on the ability of LogFlow to predict log entries.

4.2.1 Methodology. LFP is implemented in Python3, using multiple processes for parallelism[9]. To also obtain parallelism at the level of I/O operations, our parallel version of the algorithm is implemented as follows in practice. The dataset is split into N files that are processed in parallel. Each process builds private hashmaps using the algorithm described previously based on the subset of log entries it is assigned. Finally, the hashmaps of the different processes are merged to run the identification of templates.

We compare LFP to the two best log parsers according to the study presented by Zhu et al. [27]: Drain [12] and IPLoM [17]. We use the implementation of IPLoM and Drain provided by LogPAI[10] and use the parameters recommended in [27]. To understand the performance benefits that are due to the parallelism and the improvements that are more generally due to our algorithm design and implementation, we evaluate two versions of LFP: a sequential version, called `LFP-ST`, where a single thread is used to process all log entries, and a parallel version, simply called `LFP` where 40 processes are used to process log entries in parallel.

The evaluation is run on a 40-core node (two 10-core E5-2660-v3 processors with 2 hyper-threads per core), with 126GB of RAM, a 6TB HDD and with Ubuntu 16.04.3. The accuracy of the parsing algorithms is measured using the metric proposed by Zhu et al. [27] for three of the datasets (`BGL`, `Spark`, `Thunderbird`). The evaluation compares the template found by the parser for each log entry to the *ground truth* and reports the percentage of correct results. The DKRZ dataset is not considered since it is not labeled.

The execution time of the log parsers is measured from the moment the parser starts reading the log journals until the template of all log entries have been discovered. The reported execution times are average over three runs of each experiment.

4.2.2 Comparison of the parsers. Table 1 reports the execution time and the accuracy for the 4 datasets. LFP is able to parse each dataset in less than 3 minutes. The results of IPLoM are not included in the table because it fails to parse any of the datasets in less than 48 hours[11]. The performance of Drain varies. It manages to parse the BGL logs in less than 1 minutes but it is unable to parse the Thunderbird dataset in less than 48h. In general LFP is much faster than Drain and, for instance, achieves a 150× speedup on the Spark dataset and a 900× speedup on the DKRZ dataset. A detailed analysis of the performance of Drain shows that it is impacted by the total number of templates present in a dataset. Its performance gets really low on the DKRZ and the Thunderbird datatsets because they include several hundreds of different templates[12].

		Execution time			Accuracy	
Dataset	# log entries	LFP	LFP-ST	Drain	Drain	LFP
BGL	4.7M	9s	47s	50s	0.96	0.95
Spark	15M	24s	226s	3594s	0.92	0.92
Thunderbird	211M	145s	799s	>48h	0.95	0.99
DKRZ	611M	162s	2388s	41h	/	/

Table 1: Execution time and accuracy of the log parsers.

The performance of the sequential version of LFP (LFP-ST) shows that large performance improvements are obtained thanks to the design of our algorithm. On the Spark dataset for instance, a 16× speedup is observed compared to Drain. Introducing parallelism further improves the performance. Additional measurements show that LFP scales well up to 16 threads (13.2× speedup compared of `LF-ST`). After this point, the scalability is more limited, which we attribute to contention on the processor caches and on the disk.

With respect to accuracy, the results of LFP are very close to the ones of Drain. This shows that the assumption we made about the format of log entries is very often valid for this kind of systems. On the Thunderbird dataset, the results of LFP are even better than the results of Drain. The 99% accuracy achieved by LFP on this dataset is better than the accuracy of any of the 13 log parsers evaluated in [27] on the same dataset.

4.2.3 Impact of the log parser on the predictions. Although the results obtained by LFP and Drain are close in terms of accuracy, there are some variations which imply that depending on the log parser that is used, the LSTM model does not have the same set of templates to analyze for a given dataset.

We evaluated the impact of using Drain or LFP proposed in this paper on the accuracy of the predictions made by the LSTM model. For this evaluation, we used the DKRZ dataset as it is the one where we observed the more differences between the templates identified by Drain and by our parser. Running the evaluation in the case where the log entries are grouped on a per-node basis, we measured the quality of the predictions. We observe that the value of the micro-metrics are the same with the two parsers. The only small difference is that a macro-precision of *only* 94% is obtained when using LFP whereas the value was 95% using Drain.

At the end, using LFP instead of Drain has almost no impact on the quality of the predictions. We think that this can be explained by the generalization capabilities of the LSTM models that manages

[9] https://docs.python.org/3/library/multiprocessing.html
[10] https://github.com/logpai/logparser commit 7be15f3

[11] The execution time of IPLoM is not consistent with the time reported by other studies [27] using the same implementation. We are working with the authors to find and correct the issue but it does not change the conclusion about this evaluation as IPLoM has in general the same efficiency as Drain.
[12] The time complexity of Drain algorithm depends on the number of templates.

to discover correlations between log entries even if the identified templates are not exactly the same.

4.3 Discussion

To make the comparison between LFP and Drain complete, it should be mentioned that Drain is an online parser while LFP works offline. This means that in a scenario where the parser has already processed a set of log entries generated by a system and has to parse new log entries that might include new templates, Drain is able to parse only the new log entries whereas LFP has to re-process all the log entries from scratch[13]. This can be an advantage in some cases but, in the context of LogFlow, it is mostly not. Indeed, if new log templates need to be taken into account, the LSTM model needs anyway to be re-trained and this training phase is going to be more costly than the parsing. It is also important to recall that LFP is able to process more than 600M log entries in less than 3 minutes, which means that re-processing all log entries from scratch would not be very costly in most cases.

Another advantage of LFP is that the algorithm has no parameters that need to be configured by the user. On the contrary, most existing log parsers have a set of parameters that need to be tuned to obtain good results [12, 17, 27].

5 CONCLUSION

The paper presents LogFlow, a tool to help human operators during the manual analysis of system logs generated by large-scale infrastructures. For a given log entry, LogFlow is able to construct a graph of correlations pinpointing the sequences of log entries that can best explain the occurrence of the selected event. Our work demonstrates how an LSTM deep-neural network augmented with an attention module to obtain an interpretable predictive model can be used to identify correlations between log entries. We additionally propose a multi-model approach to deal with imbalanced datasets that further increases the ability of LogFlow to identify correlations. Finally, to allow LogFlow to deal with very large datasets, we propose a new log parser algorithm (LFP) based on a simple design that allows log entries to be parsed in parallel and provides a very good performance. Our evaluations with several datasets coming from various systems demonstrate the accuracy of LogFlow in the detection of correlations between logs entries and its efficiency.

ACKNOWLEDGMENTS

The work presented in this paper has been partially founded by the ITEA PAPUD project and the Paris region SYSTEMATIC hub, as well as by the national project PIA FSN HYDDA. We would also like to thank DKRZ (Deutsches Klimarechenzentrum) for giving us access to their data and helping us analyzing them.

REFERENCES

[1] Jeremy Appleyard. 2016. Optimizing Recurrent Neural Networks in cuDNN 5. https://developer.nvidia.com/blog/optimizing-recurrent-neural-networks-cudnn-5/.
[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* (2003).
[3] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Workshop on Machine Learning for Computing Systems*.
[4] Mateusz Buda, Atsuto Maki, and Maciej A Mazurowski. 2018. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks* 106 (2018).
[5] Anwesha Das, Frank Mueller, Paul Hargrove, Eric Roman, and Scott Baden. 2018. Doomsday: Predicting which node will fail when on supercomputers. In *SuperComputing'18*.
[6] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 40–51.
[7] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. 2018. LogLens: A real-time log analysis system. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.
[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.
[9] Xiaoyu Fu, Rui Ren, Sally A McKee, Jianfeng Zhan, and Ninghui Sun. 2014. Digging deeper into cluster system logs for failure prediction and root cause diagnosis. In *IEEE International Conference on Cluster Computing*.
[10] Ana Gainaru, Franck Cappello, Joshi Fullop, Stefan Trausan-Matu, and William Kramer. 2011. Adaptive event prediction strategy with dynamic time window for large-scale hpc systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*.
[11] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in large scale systems: long-term measurement, analysis, and implications. In *SuperComputing'17*.
[12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.
[13] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. arXiv:2008.06448
[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997).
[15] David Jauk, Dai Yang, and Martin Schulz. 2019. Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *SuperComputing'19*.
[16] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. arXiv:1508.04025
[17] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM international conference on Knowledge discovery and data mining*. 1255–1264.
[18] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *International Joint Conference on Artificial Intelligence*.
[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv:1301.3781* (2013).
[20] Adam Oliner and Jon Stearley. 2007. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 575–584.
[21] Wenjie Pei, Tadas Baltrusaitis, David MJ Tax, and Louis-Philippe Morency. 2017. Temporal attention-gated model for robust sequence classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6730–6739.
[22] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What can we learn from four years of data center hardware failures?. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 25–36.
[23] Shoujin Wang, Wei Liu, Jia Wu, Longbing Cao, Qinxue Meng, and Paul J Kennedy. 2016. Training deep neural networks on imbalanced data sets. In *2016 international joint conference on neural networks*. IEEE, 4368–4374.
[24] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.
[25] Min-Ling Zhang and Zhi-Hua Zhou. 2013. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering* 26, 8 (2013).
[26] Ziming Zheng, Li Yu, Zhiling Lan, and Terry Jones. 2012. 3-dimensional root cause diagnosis via co-analysis. In *Proceedings of the 9th international conference on Autonomic computing*. ACM, 181–190.
[27] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 121–130.

[13] Note however that if no new templates are included in the new log entries, LogFlow is also able to process directly the new log entries