

LogBERT: Log Anomaly Detection via BERT

Haixuan Guo¹, Shuhan Yuan¹, and Xintao Wu²

¹ Utah State University, Logan UT
ghaixan95@aggiemail.usu.edu, shuhan.yuan@usu.edu

² University of Arkansas, Fayetteville AR
xintaowu@uark.edu

Abstract. Detecting anomalous events in online computer systems is crucial to protect the systems from malicious attacks or malfunctions. System logs, which record detailed information of computational events, are widely used for system status analysis. In this paper, we propose LogBERT, a self-supervised framework for log anomaly detection based on Bidirectional Encoder Representations from Transformers (BERT). LogBERT learns the patterns of normal log sequences by two novel self-supervised training tasks and is able to detect anomalies where the underlying patterns deviate from normal log sequences. The experimental results on three log datasets show that LogBERT outperforms state-of-the-art approaches for anomaly detection.

Keywords: anomaly detection · log sequences · BERT

1 Introduction

Online computer systems are vulnerable to various malicious attacks in cyberspace. Detecting anomalous events from online computer systems in a timely manner is the fundamental step to protect the systems. System logs, which record detailed information about computational events generated by computer systems, play an important role in anomaly detection nowadays.

Currently, many traditional machine learning models are proposed for identifying anomalous events from log messages. These approaches extract useful features from log messages and adopt machine learning algorithms to analyze the log data. Due to the data imbalance issue, it is infeasible to train a binary classifier to detect anomalous log sequences. As a result, many unsupervised learning models, such as Principal Component Analysis (PCA) [19], or one class classification models, such as one-class SVM [5,16], are widely-used to detect anomalies. However, traditional machine learning models, such as one-class SVM, are hard to capture the temporal information of discrete log messages.

Recently, deep learning models, especially recurrent neural networks (RNNs), are widely used for log anomaly detection since they are able to model the sequential data [2,23,17]. However, there are still some limitations of using RNN for modeling log data. First, although RNN can capture the sequential information by the recurrence formula, it cannot make each log in a sequence encoding the

context information from both the left and right context. However, it is crucial to observe the complete context information instead of only the information from previous steps when detecting malicious attacks based on log messages. Second, current RNN-based anomaly detection models are trained to capture the patterns of normal sequences by predicting the next log message given previous log messages. This training objective mainly focuses on capturing the correlation among the log messages in normal sequences. When such correlation in a log sequence is violated, the RNN model cannot correctly predict the next log message based on previous ones. Then, we will label the sequence as anomalous. However, only using the prediction of next log message as objective function cannot not explicitly encode the common patterns shared by all normal sequences.

To tackle the existing limitations of RNN-based models, in this work, we propose LogBERT, a self-supervised framework for log anomaly detection based on Bidirectional Encoder Representations from Transformers (BERT). Inspired by the great success of BERT in modeling sequential text data [1], we leverage BERT to capture patterns of normal log sequences. By using the structure of BERT, we expect the contextual embedding of each log entry can capture the information of whole log sequences. To achieve that, we propose two self-supervised training tasks: 1) masked log key prediction, which aims to correctly predict log keys in normal log sequences that are randomly masked; 2) volume of hypersphere minimization, which aims to make the normal log sequences close to each other in the embedding space. After training, we expect LogBERT encodes the information about normal log sequences. We then derive a criterion to detect anomalous log sequences based on LogBERT. Experimental results on three log datasets show that LogBERT achieves the best performance on log anomaly detection by comparing with various state-of-the-art baselines.

2 Related Work

System logs are widely used by large online computer systems for troubleshooting, where each log message is usually a semi-structured text string. The traditional approaches explicitly use the keywords (e.g., “fail”) or regular expressions to detect anomalous log entries. However, these approaches cannot detect malicious attacks based on a sequence of operations, where each log entry looks normal, but the whole sequence is anomalous. To tackle this challenge, many rule-based approaches are proposed to identify anomalous events [11,20]. Although rule based approaches can achieve high accuracy, they can only identify pre-defined anomalous scenarios and require heavy manual engineering.

As malicious attacks become more complicated, learning-based approaches are proposed. The typical pipeline for these approaches consists of three steps [4]. First, a log parser is adopted to transform log messages to log keys. A feature extraction approach, such as TF-IDF, is then used to build a feature vector to represent a sequence of log keys in a sliding window. Finally, in most cases, an unsupervised approach is applied for detecting the anomalous sequences [18,9].

Recently, many deep learning-based log anomaly detection approaches are proposed for log anomaly detection [21,2,22,23,8,17]. Most of the existing approaches adopt recurrent neural networks, especially long-short term memory (LSTM) or gated recurrent unit (GRU) to model the normal log key sequences and derive anomalous scores to detect the anomalous log sequences [2,23,17]. In this work, we explore the advanced BERT model to capture the information of log sequences and propose two novel self-supervised tasks to train the model.

3 LogBERT

In this section, we introduce our framework, LogBERT, for log sequence anomaly detection. Inspired by BERT [1], LogBERT leverages the Transformer encoder to model log sequences and is trained by novel self-supervised tasks to capture the patterns of normal sequences. Figure 1 shows the whole framework of LogBERT.

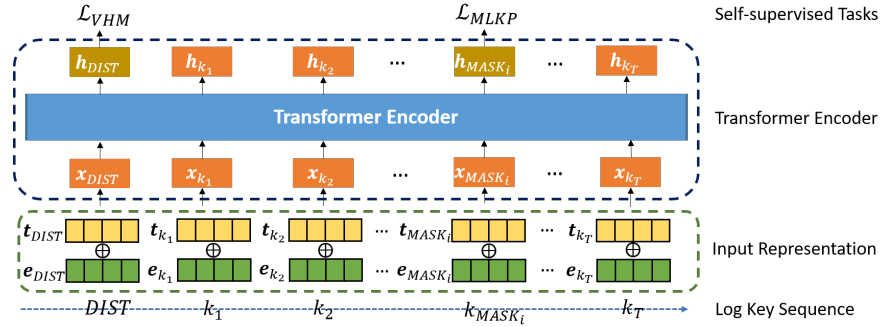


Fig. 1: The overview of LogBERT

3.1 Framework

Given a sequence of unstructured log messages, we aim to detect whether this sequence is normal or anomalous. In order to represent log messages, following a typical pre-processing approach, we first extract log keys (string templates) from log messages via a log parser (shown in Figure 2). Then, we can define a log sequence as a sequence of ordered log keys $S = \{k_1, \dots, k_t, \dots, k_T\}$, where $k_t \in \mathcal{K}$ indicates the log key in the t -th position, and $\mathcal{K}$ indicates a set of log keys extracted from log messages. The goal of this task is to predict whether a new log sequence S is anomalous based on a training dataset $\mathcal{D} = \{S^j\}_{j=1}^N$ that consists of only normal log sequences. To achieve that, LogBERT models the normal sequences and further derive an anomaly detection criterion to identify anomalous sequences.

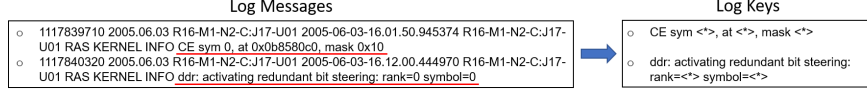


Fig. 2: Log messages in the BGL dataset and the corresponding log keys extracted by a log parser. The message with red underscore indicates the detailed computational event.

Input Representation. Given a normal log sequence S^j , we first add a special token, DIST, at the beginning of $S^j = \{k_1^j, \dots, k_t^j, \dots, k_T^j\}$ as the first log key, which is used to represent the whole log sequence based on the structure of Transformer encoder. LogBERT then represents each log key k_t^j as an input representation $\mathbf{x}_t^j$, where the representation $\mathbf{x}_t^j$ is a summation of a log key embedding and a position embedding. In this work, we randomly generate a matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{K}|*d}$ as the log key embedding matrix, where d is the dimension of log key embedding, while the position embeddings $\mathbf{T} \in \mathbb{R}^{T*d}$ are generated by using a sinusoid function to encode the position information of log keys in a sequence [1]. Finally, the input representation of the log key k_t is defined as: $\mathbf{x}_t^j = \mathbf{e}_{k_t^j} + \mathbf{t}_{k_t^j}$.

Transformer Encoder. LogBERT adopts Transformer encoder to learn the contextual relations among log keys in a sequence. Transformer encoder consists of multiple transformer layers. Each transformer layer includes a multi-head self-attention and a position-wise feed forward sub-layer in which a residual connection is employed around each of two sub-layers, followed by layer normalization [15]. The multi-head attention employs H parallel self-attentions to jointly capture different aspect information at different positions over the input log sequence. Formally, for the l -th head of the attention layer, the scaled dot-product self-attention is defined as:

$$head_l = Attention(\mathbf{X}^j \mathbf{W}_l^Q, \mathbf{X}^j \mathbf{W}_l^K, \mathbf{X}^j \mathbf{W}_l^V), \quad (1)$$

where $Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{QK}^T}{\sqrt{d_v}})\mathbf{V}$; $\mathbf{X}^j \in \mathbb{R}^{T*d}$ is the input representation of the log sequence; $\mathbf{W}_l^Q$, $\mathbf{W}_l^K$ and $\mathbf{W}_l^V$ are linear projection weights with dimensions $\mathbb{R}^{d*d_v}$ for the l -th head, and d_v is the dimension for one head of the attention layer. Each self-attention makes each key attend to all the log keys in an input sequence and computes the hidden representation for each log key with an attention distribution over the sequence.

The multi-head attention employs a parallel of self-attentions to jointly capture different aspect information at different log keys. Formally, the multi-head attention concatenates H parallel heads together as:

$$f(\mathbf{X}^j) = Concat(head_1, \dots, head_H) \mathbf{W}^O, \quad (2)$$

where $\mathbf{W}^O \in \mathbb{R}^{hd_v*d_o}$ is a projection matrix, and d_o is the dimension for the output of multi-head attention sub-layer.

Then, the position-wise feed forward sub-layer with a ReLU activation is applied to the hidden representation of each activity separately. Finally, by combining the position-wise feed forward sub-layer and multi-head attention, a transformer layer is defined as:

$$\text{transformer_layer}(\mathbf{X}^j) = FFN(f(\mathbf{X}^j)) = ReLU(f(\mathbf{X}^j)\mathbf{W}_1)\mathbf{W}_2, \quad (3)$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are trained projection matrices.

The Transformer encoder usually consists of multiple transformer layers. We denote $\mathbf{h}_t^j$ as the contextual embedding vector of the log key k_t^j produced by the Transformer encoder, i.e., $\mathbf{h}_t^j = \text{Transformer}(x_t^j)$.

3.2 Objective Function

In order to train the LogBERT model, we propose two self-supervised training tasks to capture the patterns of normal log sequences.

Task I: Masked Log Key Prediction (MLKP). In order to capture the bidirectional context information of log sequences, we train LogBERT to predict the masked log keys in log sequences. In our scenario, LogBERT takes log sequences with random masks as inputs, where we randomly replace a ratio of log keys in a sequence with a specific MASK token. The training objective is to accurately predict the randomly masked log keys. The purpose is to make LogBERT encode the prior knowledge of normal log sequences.

To achieve that, we feed the contextual embedding vector of the i -th MASK token in the j -th log sequence $\mathbf{h}_{[\text{MASK}_i]}^j$ to a softmax function, which will output a probability distribution over the entire set of log keys $\mathcal{K}$:

$$\hat{\mathbf{y}}_{[\text{MASK}_i]}^j = \text{Softmax}(\mathbf{W}_C \mathbf{h}_{[\text{MASK}_i]}^j + \mathbf{b}_C), \quad (4)$$

where $\mathbf{W}_C$ and $\mathbf{b}_C$ are trainable parameters. Then, we adopt the cross entropy loss as the objective function for masked log key prediction, which is defined as:

$$\mathcal{L}_{MLKP} = -\frac{1}{N} \sum_{j=1}^N \sum_{i=1}^M \mathbf{y}_{[\text{MASK}_i]}^j \log \hat{\mathbf{y}}_{[\text{MASK}_i]}^j, \quad (5)$$

where $\mathbf{y}_{[\text{MASK}_i]}^j$ indicates the real log key for the i -th masked token, and M is the total number of masked tokens in the j -th log sequence. Since the patterns of normal and anomalous log sequences are different, we expect once LogBERT is able to correctly predict the masked log keys, it can distinguish the normal and anomalous log sequences.

Task II: Volume of Hypersphere Minimization (VHM). Inspired by the Deep SVDD approach [13], where the objective is to minimize the volume of a data-enclosing hypersphere, we propose a spherical objective function to regulate the distribution of normal log sequences. The motivation is that normal log sequences should be concentrated and close to each other in the embedding space, while the anomalous log sequences are far to the center of the sphere. We

first derive the representations of normal log sequences and then compute the center representation based on the mean operation. In particular, we consider the contextual embedding vector of the DIST token $\mathbf{h}_{\text{DIST}}^j$, which encodes the information of entire log sequence based on the Transformer encoder, as the representation of a log sequence in the embedding space. To make the representations of normal log sequences close to each other, we further derive the center representation of normal log sequences $\mathbf{c}$ in the training set by a mean operation, i.e., $\mathbf{c} = \text{Mean}(\mathbf{h}_{\text{DIST}}^j)$. Then, the objective function is to make the representation of normal log sequence $\mathbf{h}_{\text{DIST}}^j$ close to the center representation $\mathbf{c}$:

$$\mathcal{L}_{VHM} = \frac{1}{N} \sum_{j=1}^N \|\mathbf{h}_{\text{DIST}}^j - \mathbf{c}\|^2. \quad (6)$$

By minimizing the Equation 6, we expect all the normal log sequences in the training set are close to the center, while the anomalous log sequences have a larger distance to the center. Meanwhile, another advantage of the spherical objective function is that by making the sequence representations close to the center, the Transformer encoder can also leverage the information from other log sequences via the center representation $\mathbf{c}$, since $\mathbf{c}$ encodes all the information of normal log sequences. As a result, the model should be able to predict the masked log keys with higher accuracy for normal log sequences because the normal log sequences should share similar patterns.

Finally, the objective function for training the LogBERT is defined as below:

$$\mathcal{L} = \mathcal{L}_{MLKP} + \alpha \mathcal{L}_{VHM}, \quad (7)$$

where α is a hyper-parameter to balance two training tasks.

3.3 Anomaly Detection

After training, we can deploy LogBERT for anomalous log sequence detection. The idea of applying LogBERT for log anomaly detection is that since LogBERT is trained on normal log sequences, it can achieve high prediction accuracy on predicting the masked log keys if a testing log sequence is normal. Hence, we can derive the anomalous score of a log sequence based on the prediction results on the MASK tokens. To this end, given a testing log sequence, similar to the training process, we first randomly replace a ratio log keys with MASK tokens and use the randomly-masked log sequence as an input to LogBERT. Then, given a MASK token, the probability distribution calculated based on Equation 4 indicates the likelihood of a log key appeared in the position of the MASK token. Similar to the strategy in DeepLog[2], we build a candidate set consisting of g normal log keys with the top g highest likelihoods computed by $\hat{\mathbf{y}}_{[\text{MASK}_i]}$. If the real log key is in the candidate set, we treat the key as normal. In other words, if the observed log key is not in the top- g candidate set predicted by LogBERT, we consider the log key as an anomalous log key. Then, when a log sequence consists of more than r anomalous log keys, we will label this log sequence as

anomalous. Both g and r are hyper-parameters and will be tuned based on a validation set.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate the proposed LogBERT on three log datasets, HDFS, BGL, and Thunderbird. Table 1 shows the statistics of the datasets. For all datasets, we adopt around 5000 normal log sequences for training. The number in the brackets under the column “# Log Keys” indicates the number of unique log keys in the training dataset.

Table 1: Statistics of evaluation datasets

Dataset	# Log Messages	# Anomalies	# Log Keys	# of Log Sequences in Test Dataset	
				Normal	Anomalous
HDFS	11,172,157	284,818	46 (15)	553,366	10,647
BGL	4,747,963	348,460	334 (175)	10,045	2,630
Thunderbird-mini	20,000,000	758,562	1,165 (866)	71,155	45,385

- Hadoop Distributed File System (HDFS) [18]. HDFS dataset is generated by running Hadoop-based map-reduce jobs on Amazon EC2 nodes and manually labeled through handcrafted rules to identify anomalies. HDFS dataset consists of 11,172,157 log messages, of which 284,818 are anomalous. For HDFS, we group log keys into log sequences based on the session ID in each log message. The average length of log sequences is 19.
- BlueGene/L Supercomputer System (BGL) [10]. BGL dataset is collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL). Logs contain alert and non-alert messages identified by alert category tags. The alert messages are considered as anomalous. BGL dataset consists of 4,747,963 log messages, of which 348,460 are anomalous. For BGL, we define a time sliding window as 5 minutes to generate log sequences, where the average length is 562.
- Thunderbird [10]. Thunderbird dataset is another large log dataset collected from a supercomputer system. We select the first 20,000,000 log messages from the original Thunderbird dataset to compose our dataset, of which 758,562 are anomalous. For Thunderbird, we also adopt a time sliding window as 1 minute to generate log sequences, where the average length is 326.

Baselines. We compare our LogBERT model with the following baselines.

- Principal Component Analysis (PCA) [19]. PCA builds counting matrix based on the frequency of log keys sequences and then reduces the original counting matrix into a low dimensional space to detect anomalous sequences.

- One-Class SVM (OCSVM) [14]. One-Class SVM is a well-known one-class classification model and widely used for log anomaly detection [5,16] by only observing the normal data.
- IsolationForest (iForest) [7]. Isolation forest is an unsupervised learning algorithm for anomaly detection by representing features as tree structures.
- LogCluster [6]. LogCluster is a clustering based approach, where the anomalous log sequences are detected by comparing with the existing clusters.
- DeepLog [2]. DeepLog is a state-of-the-art log anomaly detection approach. DeepLog adopts recurrent neural network to capture patterns of normal log sequences and further identifies the anomalous log sequences based on the performance of log key predictions.
- LogAnomaly [23]. Log Anomaly is a deep learning-based anomaly detection approach and able to detect sequential and quantitative log anomalies.

Implementation Details. We adopt Drain [3] to parse the log messages into log keys. Regarding baselines, we leverage the package *Loglizer* [4] to evaluate PCA, OCSVM, iForest as well as LogCluster for anomaly detection and adopt the open source deep learning-based log analysis toolkit to evaluate DeepLog and LogAnomaly ³. For LogBERT, we construct a Transformer encoder by using two Transformer layers. The dimensions for the input representation and hidden vectors are 50 and 256, respectively. The hyper-parameters, including α in Equation 7, m the ratio of masked log keys for the MKLP task, r the number of predicted anomalous log keys, and g the size of top- g candidate set for anomaly detection are tuned based on a small validation set. In our experiments, both training and detection phases have the same ratio of masked log keys m . The code of LogBERT is available online ⁴.

4.2 Experimental Results

Table 2: Experimental Results on HDFS, BGL, and Thunderbird Datasets

Method	HDFS			BGL			Thunderbird		
	Precision	Recall	F-1 score	Precision	Recall	F-1 score	Precision	Recall	F-1 score
PCA	5.89	100.00	11.12	9.07	98.23	16.61	37.35	100.00	54.39
iForest	53.60	69.41	60.49	99.70	18.11	30.65	34.45	1.68	3.20
OCSVM	2.54	100.00	4.95	1.06	12.24	1.96	18.89	39.11	25.48
LogCluster	99.26	37.08	53.99	95.46	64.01	76.63	98.28	42.78	59.61
DeepLog	88.44	69.49	77.34	89.74	82.78	86.12	87.34	99.61	93.08
LogAnomaly	94.15	40.47	56.19	73.12	76.09	74.08	86.72	99.63	92.73
LogBERT	87.02	78.10	82.32	89.40	92.32	90.83	96.75	96.52	96.64

Performance on Log Anomaly Detection. Table 2 shows the results of LogBERT as well as baselines on three datasets. We can notice that PCA, Isolation Forest, and OCSVM have poor performance on log anomaly detection.

³ <https://github.com/donglee-afar/logdeep>

⁴ <https://github.com/HelenGuohx/logbert>

Although these methods could achieve extremely high precision or recall values, they cannot balance the performance on both precision and recall, which lead to extremely low F1 scores. This could be because using the counting vector to represent a log sequence leads to the loss of temporal information from sequences. LogCluster, which is designed for log anomaly detection, achieves better performance than the PCA, Isolation Forest, and OCSVM. Meanwhile, two deep learning-based baselines, DeepLog and LogAnomaly, significantly outperform the traditional approaches and achieve reasonable F1 scores on three datasets, which show the advantage to adopt deep learning models to capture the patterns of log sequences. Moreover, our proposed LogBERT achieves the highest F1 scores on three datasets with large margins by comparing with all baselines. It indicates that by using self-supervised training tasks, LogBERT can successfully model the normal log sequences and further identify anomalous sequences with high accuracy.

Table 3: Performance of LogBERT base on One Self-supervised Training Task

	HDFS			BGL			Thunderbird		
	Precision	Recall	F-1 score	Precision	Recall	F-1 score	Precision	Recall	F-1 score
MLKP	77.54	78.65	78.09	93.16	86.46	89.69	97.07	95.90	96.48
VHM	2.43	39.17	4.58	71.04	43.84	54.22	56.58	43.87	49.42
Both	87.02	78.10	82.32	89.40	92.32	90.83	96.75	96.52	96.64

Ablation Studies. In order to further understand our proposed LogBERT, we conduct ablation experiments on three log datasets. LogBERT is trained by two self-supervised tasks. We evaluate the performance of LogBERT by only using one training task each time. When the model is only trained by minimizing the volume of hypersphere, we identify anomalous log sequences by computing distances of the log sequence representations to the center of normal log sequences $\mathbf{c}$. If the distance is larger than a threshold, we consider a log sequence is anomalous. Table 3 shows the experimental results. We can notice that when only using the task of masked log key prediction to train the model, we can still get very good performance on log anomaly detection, which shows the effectiveness of training the model by predicting masked log keys. We can also notice that even we do not train the LogBERT with the task of the volume of hypersphere minimization, LogBERT achieves higher F1 scores than DeepLog on all three datasets, which shows that compared with LSTM, Transformer encoder is better at capturing the patterns of log sequences. Meanwhile, we can observe that when only training the model for minimizing the volume of hypersphere, the performance is poor. It indicates that only using distance as a measure to identify anomalous log sequences cannot achieve good performance. However, combining two self-supervised tasks to train LogBERT can achieve better performance than the models only trained by one task. Especially, for the HDFS dataset, LogBERT trained by two self-supervised tasks gains a large margin in terms of F1 score (82.32) compared with the model only trained by MLKP (78.09). For BGL and

Thunderbird, the improvement of LogBERT is not as significant as the model in HDFS. This could be because the average length of log sequences in BGL (562) and Thunderbird (326) datasets are much larger than the log sequences in HDFS (19). For longer sequences, only predicting the masked log keys can capture the most important patterns of log sequences since there are many more mask tokens in longer sequences. On the other hand, for short log sequences, we cannot have many masks tokens. As a result, the task of the volume of hypersphere minimization can help to boost the performance. Hence, based on Table 3, we can conclude that using two self-supervised tasks to train LogBERT can achieve better performance, especially when the log sequences are relatively short.

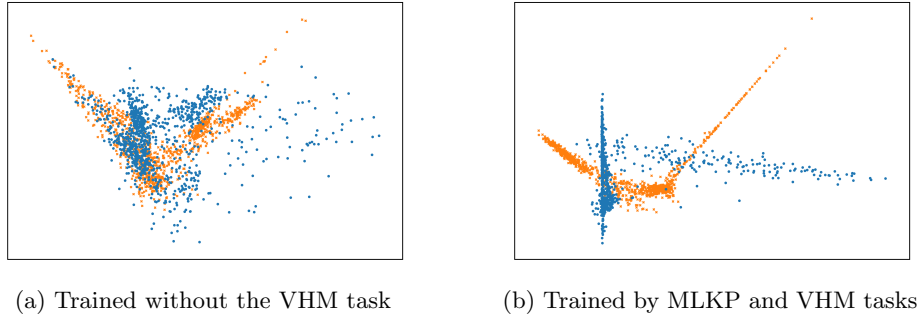


Fig. 3: Visualization of log sequences by using the contextual embedding of DIST tokens $\mathbf{h}_{\text{DIST}}$. The blue dots indicate the normal log sequences, while the orange ‘x’ symbols indicate anomalous log sequences.

Visualization. In order to visualize the log sequences, we adopt locally linear embedding (LLE) algorithm [12] to map the log sequence representations into a two dimensional space, where the contextual embedding of DIST token $\mathbf{h}_{\text{DIST}}$ is used as the representation of a log sequence. We randomly select 1000 normal and 1000 anomalous sequences from the HDFS dataset for visualization. Figure 3 shows the visualization results of log sequences trained by LogBERT with and without the VHM task. We can notice that the normal and anomalous log sequences are mixed together when we trained the model without the VHM task (shown in Figure 3a). On the contrary, as shown in Figure 3b, by incorporating the VHM task, the normal and anomalous log sequences are clearly separated in the latent space, and the normal log sequences group together. Therefore, the visualization presents that the VHM task is effective in regulating the model to split the normal and abnormal data in latent space.

Parameter analysis. We analyze the sensitivity of model performance by tuning various hyper-parameters. Figure 4a shows that the model performance is relatively stable by setting different α values in Equation 7. This is because, for the BGL dataset, the loss from the masked log key prediction dominates the final loss value due to the long log sequences. As a result, the weight for the VHM

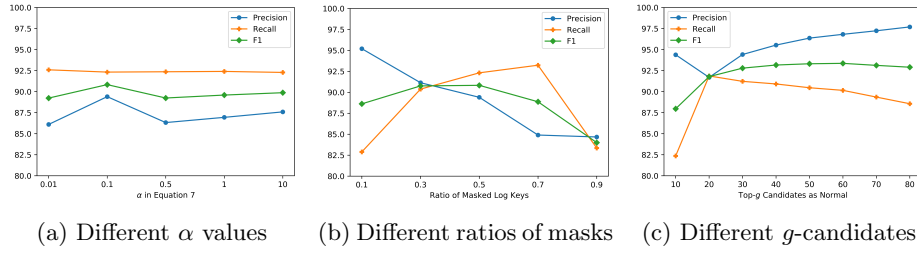


Fig. 4: Parameter analysis on the BGL dataset.

task does not have much influence on the performance. Figure 4b shows the performance with different ratios of masked log keys. Note that we use the same ratio in both training and detection phases. We can notice that increasing the ratios of masked log keys in the sequences from 0.1 to 0.5 can slightly increase the F1 scores while keeping increasing the ratios makes the performance worse. This is because while the masked log keys increase in a reasonable range, the model can capture more information about the sequence. However, if a sequence contains too many masked log keys, it loses too much information for making the predictions. Figure 4c shows that when increasing the size of the candidate set as normal log keys, the precision for anomaly detection keeps increasing while the recall is reducing, which meets our expectation. Hence, we need to find the appropriate size of the candidate set to balance the precision and recall for the anomaly detection.

5 Conclusion

Log anomaly detection is essential to protect online computer systems from malicious attacks or malfunctions. In this paper, we have developed LogBERT, a novel log anomaly detection model based on BERT. In order to train LogBERT only based on normal log sequences, we have proposed two self-supervised training tasks. One is to predict the masked log keys in log sequences, while the other is to make the normal log sequences close to each other in the embedding space. After training over normal log sequences, LogBERT is able to detect anomalous log sequences. Experimental results on three log datasets have shown that LogBERT outperforms the state-of-the-art approaches for log anomaly detection.

References

1. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs] (2018)
2. Du, M., Li, F., Zheng, G., Srikumar, V.: DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. (2017).

3. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: An Online Log Parsing Approach with Fixed Depth Tree. In: 2017 IEEE International Conference on Web Services (ICWS). (2017).
4. He, S., Zhu, J., He, P., Lyu, M.R.: Experience Report: System Log Analysis for Anomaly Detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). (2016).
5. Li, K.L., Huang, H.K., Tian, S.F., Xu, W.: Improving one-class SVM for anomaly detection. In: Proceedings of the 2003 International Conference on Machine Learning and Cybernetics. (2003).
6. Lin, Q., Zhang, H., Lou, J., Zhang, Y., Chen, X.: Log Clustering Based Problem Identification for Online Service Systems. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (2016)
7. Liu, F.T., Ting, K.M., Zhou, Z.: Isolation Forest. In: 2008 Eighth IEEE International Conference on Data Mining. pp. 413–422 (2008).
8. Liu, F., Wen, Y., Zhang, D., Jiang, X., Xing, X., Meng, D.: Log2vec: A Heterogeneous Graph Embedding Based Approach for Detecting Cyber Threats within Enterprise. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. (2019).
9. Lou, J.G., Fu, Q., Yang, S., Xu, Y., Li, J.: Mining invariants from console logs for system problem detection. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. (2010)
10. Oliner, A., Stearley, J.: What supercomputers say: A study of five system logs. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). pp. 575–584. IEEE (2007)
11. Pecchia, A., Cinque, M., Cotroneo, D.: Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering* **39**(06), 806–821 (2013).
12. Roweis, S.T., Saul, L.K.: Nonlinear dimensionality reduction by locally linear embedding. *science* **290**(5500), 2323–2326 (2000)
13. Ruff, L., Vandermeulen, R., Goernitz, N., Deecke, L., Siddiqui, S.A., Binder, A., Müller, E., Kloft, M.: Deep One-Class Classification. In: International Conference on Machine Learning. pp. 4393–4402. PMLR (2018)
14. Schölkopf, B., Platt, J.C., Shawe-Taylor, J., Smola, A.J., Williamson, R.C.: Estimating the Support of a High-Dimensional Distribution. *Neural Computation* **13**(7), 1443–1471 (2001).
15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention Is All You Need. *arXiv:1706.03762 [cs]* (2017)
16. Wang, Y., Wong, J., Miner, A.: Anomaly intrusion detection using one class SVM. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004. pp. 358–364 (2004).
17. Wang, Z., Chen, Z., Ni, J., Liu, H., Chen, H., Tang, J.: Multi-Scale One-Class Recurrent Neural Networks for Discrete Event Sequence Anomaly Detection. In: WSDM (2021)
18. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.: Online system problem detection by mining patterns of console logs. In: 2009 Ninth IEEE International Conference on Data Mining. pp. 588–597. IEEE (2009)
19. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. (2009).

20. Yen, T.F., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W., Juels, A., Kirda, E.: Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp. 199–208 (2013)
21. Zhang, K., Xu, J., Min, M.R., Jiang, G., Pelechrinis, K., Zhang, H.: Automated IT system failure prediction: A deep learning approach. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 1291–1300 (2016).
22. Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., Chen, J., He, X., Yao, R., Lou, J.G., Chintalapati, M., Shen, F., Zhang, D.: Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. (2019).
23. Zhou, R., Sun, P., Tao, S., Zhang, R., Meng, W., Liu, Y., Zhu, Y., Liu, Y., Pei, D., Zhang, S., Chen, Y.: LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In: IJCAI. pp. 4739–4745 (2019)
24. He, S., Zhu, J., He, P., Lyu, M.R.: Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. arXiv:2008.06448 [cs] (Aug 2020)

LogLLM: Log-based Anomaly Detection Using Large Language Models

Wei Guan¹, Jian Cao^{1*}, Shiyu Qian¹, Jianqi Gao¹, Chun Ouyang²

¹*Department of Computer Science and Engineering, SJTU, Shanghai, China*

²*The School of Information Systems, QUT, Brisbane, Australia*

{guan-wei, cao-jian, qshiyu, 193139}@sjtu.edu.cn, c.ouyang@qut.edu.au

Abstract—Software systems often record important runtime information in logs to help with troubleshooting. Log-based anomaly detection has become a key research area that aims to identify system issues through log data, ultimately enhancing the reliability of software systems. Traditional deep learning methods often struggle to capture the semantic information embedded in log data, which is typically organized in natural language. In this paper, we propose LogLLM, a log-based anomaly detection framework that leverages large language models (LLMs). LogLLM employs BERT for extracting semantic vectors from log messages, while utilizing Llama, a transformer decoder-based model, for classifying log sequences. Additionally, we introduce a projector to align the vector representation spaces of BERT and Llama, ensuring a cohesive understanding of log semantics. Unlike conventional methods that require log parsers to extract templates, LogLLM preprocesses log messages with regular expressions, streamlining the entire process. Our framework is trained through a novel three-stage procedure designed to enhance performance and adaptability. Experimental results across four public datasets demonstrate that LogLLM outperforms state-of-the-art methods. Even when handling unstable logs, it effectively captures the semantic meaning of log messages and detects anomalies accurately.

Index Terms—System log, anomaly detection, large language model, deep learning, log analysis

I. INTRODUCTION

Ensuring high availability and reliability is crucial for large-scale software-intensive systems [1], [2]. As these systems become more complex and expansive, the occurrence of anomalies becomes unavoidable [3], [4]. Even a minor issue can lead to performance degradation, data integrity problems, and substantial losses in both customers and revenue. Therefore, anomaly detection is vital for maintaining the health and stability of complex software-intensive systems [5].

Software-intensive systems typically produce console logs that record system states and critical runtime events [6]. Engineers can utilize this log data to evaluate system health, identify anomalies, and trace the root causes of issues. However, due to the potentially vast volume of logs, manually analyzing them for anomalies can be both labor-intensive and prone to mistakes [7]. Consequently, log-based anomaly detection has emerged as a key area in automated log analysis, focusing on the automatic identification of system anomalies through log data.

Numerous deep learning-based methods [8]–[22] for log-based anomaly detection have been proposed. These methods typically employ sequential deep learning models such as LSTM [23] and transformers [24]. These methods can be further divided into reconstruction-based methods [8]–[15] and binary classification-based methods [16]–[22]. Reconstruction-based methods involve designing and training a deep neural network to reconstruct input log sequences, with anomalies detected based on reconstruction errors. The underlying principle is that anomalous samples cannot be accurately reconstructed. Binary classification-based methods, on the other hand, involve designing a binary classifier to classify samples as either normal or anomalous. These methods often require labeled anomalies for training purposes. It is recognized that system logs are documented in natural language and contain a significant amount of semantic information. Nevertheless, traditional deep learning-based methods struggle to effectively capture this information.

In recent years, significant advancements have been achieved in LLMs, such as GPT-4 [25], Llama 3 [26], and ChatGLM [27]. These models are characterized by their vast parameter sizes and are pretrained on substantially larger datasets, ranging from several gigabytes to terabytes in size. This extensive pretraining equips them with remarkable language comprehension abilities, enabling superior performance in tasks such as summarization, paraphrasing, and instruction following even in zero-shot scenarios [28]. Existing methods that utilize LLMs for log-based anomaly detection can be categorized into prompt engineering-based [7], [29]–[31] and fine-tuning-based [3], [32]–[40] approaches. Prompt engineering-based methods leverage the zero/few-shot capabilities of LLMs to detect anomalies based solely on the models’ internal knowledge. However, these methods often struggle to customize solutions for specific datasets, leading to suboptimal detection performance. Fine-tuning-based methods integrate LLMs into deep neural networks and tailor them to user-specific datasets. Nevertheless, these methods encounter challenges such as limited semantic understanding, suboptimal LLM utilization (relying solely on LLMs for semantic information extraction), and insufficient consideration of input data format, which can lead to memory overflow.

To tackle the aforementioned challenges, we propose LogLLM, a novel log-based anomaly detection framework that harnesses LLMs. Unlike traditional methods that rely on

*Corresponding author.

log parsers for template extraction, LogLLM preprocesses log messages using regular expressions, thereby streamlining the entire process. LogLLM, a fine-tuning-based method, utilizes BERT, a transformer encoder-based model, to extract semantic vectors from log messages. Additionally, it employs Llama, a transformer decoder-based model, to classify log sequences. To ensure coherence in log semantics, we introduce a projector that aligns the vector representation spaces of BERT and Llama. Our framework is trained using a novel three-stage procedure designed to enhance both performance and adaptability.

As illustrated in Section V-G, LLMs frequently face out-of-memory challenges due to their extensive parameter sizes [41]. Directly inputting the entire log sequence (by concatenating log messages into a long string) into Llama can lead to out-of-memory issues and potentially confuse the LLM, making it difficult to focus on key points for distinguishing anomalies. By adopting BERT to summarize each log message, LogLLM effectively mitigates these problems. We conduct experiments across four public datasets, and the results demonstrate that LogLLM outperforms state-of-the-art methods. Even when handling unstable logs, where new log templates frequently emerge due to software evolution, it effectively captures the semantic meaning of log messages and detects anomalies accurately. The ablation study confirms the effectiveness of the three-stage training procedure.

The main contributions of our work are as follows:

- We introduce LogLLM, a novel log-based anomaly detection framework leveraging LLMs. This study marks the first attempt to simultaneously employ transformer encoder-based and decoder-based LLMs, specifically BERT and Llama, for log-based anomaly detection.
- We propose a novel three-stage procedure to optimize the training and coordination of different components within the deep model, enhancing both performance and adaptability.
- We conduct extensive experiments on four publicly available real-world datasets, demonstrating that LogLLM achieves exceptional performance.

II. RELATED WORK

In this section, we explore related work in the field of log-based anomaly detection, with a particular focus on deep learning-based methods. We give special attention to approaches that utilize pretrained LLMs.

A. Traditional Deep Learning for Log-based Anomaly Detection

Many traditional deep learning-based methods for log-based anomaly detection have been proposed. These works can be grouped into two types based on the training paradigm: reconstruction-based methods and binary classification-based methods.

Reconstruction-based methods [8]–[15] involve designing and training a deep neural network to reconstruct input log

sequences. Anomalies are detected based on reconstruction errors. Normal log sequences can be reconstructed with minimal errors, while anomalous log sequences cannot be effectively reconstructed, resulting in significantly higher reconstruction errors. These methods consistently train the deep model on normal data that is free of anomalies, which means they are semi-supervised.

DeepLog [8] adopts LSTM to predict the next log template ID based on past log sequences. Similarly, LogAnomaly [9] predicts the next log template ID based on both sequential and quantitative patterns. Autoencoders (AEs) [10]–[13] and generative adversarial networks (GANs) [14], [15] are widely used in reconstruction-based methods. For example, LogAttn [10] adopts an AE that incorporates a temporal convolutional network (TCN) to capture temporal semantic correlations and a deep neural network (DNN) to capture statistical correlations. Duan et al. [14] use a GAN, where an encoder-decoder framework based on LSTM serves as the generator. Convolutional neural networks (CNNs) are used as the discriminator. The reconstruction error is calculated based on the difference between the input and the output from the generator.

Binary classification-based methods [16]–[22] often employ deep neural networks that output either one or two values. Typically, a single value represents the probability that a sample belongs to the anomalous class, and anomalies are detected by applying a threshold to convert this probability into a binary classification. When two values are output, they represent the probabilities of the sample belonging to the normal and anomalous classes, respectively.

Most methods [16]–[20] typically train deep models in a supervised manner. For example, Zhang et al. [16] propose LayerLog, which integrates word, log, and logseq layers to extract semantic features from log sequences. CNNs are utilized in [17], [18] to develop a binary classifier. LogRobust [19] integrates a pre-trained Word2Vec model, specifically FastText [42], and combines it with TF-IDF weights to learn representation vectors of log templates. These vectors are then fed into an attention-based Bi-LSTM model for anomaly detection. LogGD [20] transforms log sequences into graphs and utilizes a graph transformer neural network that combines graph structure and node semantics for log-based anomaly detection.

Some work [21], [22] involves training binary classifiers in a semi-supervised manner. For example, Trine [21] uses a transformer encoder [24] to encode normal log sequences into vector representations and a generator to produce random fake vector representations. The discriminator, which is composed of a transformer and a multi-layer perceptron (MLP), is trained to distinguish whether the given vector representations are normal log sequences and it is subsequently used to detect anomalies. PLELog [22] tackles the challenge of insufficient labeling by employing probabilistic label estimation and develops an attention-based GRU neural network for anomaly detection.

It is acknowledged that system logs are recorded in natural

Header	Content
1117838978 2005.06.03 R02-M1-N0-C:J12-U11 2005-06-03-15.49.38.026704 R02-M1-N0-C:J12-U11 RAS KERNEL INFO	instruction cache parity error corrected
1117843015 2005.06.03 R21-M1-N6-C:J08-U11 2005-06-03-16.56.55.309974 R21-M1-N6-C:J08-U11 RAS KERNEL INFO	141 double-hammer alignment exceptions
1117848119 2005.06.03 R16-M1-N2-C:J17-U01 2005-06-03-18.21.59.871925 R16-M1-N2-C:J17-U01 RAS KERNEL INFO	CE sym 2, at 0x0b85eee0, mask 0x05
...	...

Fig. 1: An example of a system log.

language and contain a substantial amount of semantic information. However, traditional deep learning-based methods face challenges in capturing this information.

B. LLMs for Log-based Anomaly Detection

Existing LLMs can be categorized into transformer encoder-based models, such as BERT [43], RoBERTa [44], and SpanBERT [45], and transformer decoder-based models, including GPT-4 [25], Llama 3 [26], and ChatGLM [27]. Two prevalent strategies for utilizing LLMs are prompt engineering and fine-tuning.

Prompt engineering-based methods [7], [29]–[31] detect anomalies solely by relying on the internal knowledge of LLMs. These methods typically employ transformer decoder-based models. For instance, Qi et al. [7] employ ChatGPT for zero-shot and few-shot log-based anomaly detection, utilizing prompt templates that integrate the log sequence directly. However, this approach becomes impractical when using a large window size for grouping log messages. Egersdoerfer et al. [30] address this issue by maintaining a summary-based memory, which summarizes the previous log messages, eliminating the need to input the entire log sequence for anomaly detection. RAGLog [31] uses a retrieval augmented generative (RAG) framework [46] to analyze log entries by querying its store of samples of normal log entries. They design prompt templates for LLMs to determine whether a queried log entry is normal or abnormal. Prompt engineering-based methods often struggle to customize solutions for specific datasets, which can lead to suboptimal detection performance in particular datasets.

Fine-tuning-based methods [3], [32]–[40] incorporate LLMs into deep neural networks and customize them to the user’s own dataset. Some methods [32]–[35], although adopting transformer encoder-based LLMs for anomaly detection, do not capture the semantic information within log sequences. For example, LogBERT [32] and LAnoBERT [33] utilize BERT to reconstruct the input sequence of log template IDs (IDs of log string templates) and detect anomalies based on reconstruction errors, disregarding the semantic information. Other methods [3], [36]–[39] use transformer encoder-based LLMs solely for extracting semantic information from log messages and then employ either smaller models [3], [36]–[38] or distance-based comparison [39] for classification. For instance, NeuralLog [3] leverages BERT to extract semantic vectors from raw log messages, which are subsequently used to detect anomalies via a transformer-based classification model. Similarly, RAPID [39] utilizes transformer encoder-based models to extract semantic vectors and performs anomaly

detection by comparing each query log sequence with its nearest document log sequence. Hadadi et al. [40] directly input template sequences parsed from log sequences, into GPT models and fine-tune it to accurately predict sequence labels. However, this approach faces two key challenges. First, the boundaries between templates within the sequences are unclear, making it difficult for the model to learn the sequential dependencies. Second, each template may be tokenized into multiple tokens by the LLM’s tokenizer, and a single sequence can contain numerous log templates. As a result, an excessive number of tokens may be generated, often exceeding the token (memory) limits of LLMs [41], thereby restricting the length of sequences that can be processed. These two challenges are further demonstrated in Section V-G.

LogLLM is a fine-tuning-based method that utilizes BERT for extracting semantic vectors from log messages and Llama, a transformer decoder-based model, for log sequence classification. This method aligns the vector representation spaces of BERT and Llama using a projector. The use of BERT ensures clear boundaries between log messages, as each message is represented by a distinct embedding vector, thereby enhancing classification performance. Moreover, when memory and parameter size of Llama are held constant, this approach can handle longer sequences compared to directly tokenizing the entire log sequence using Llama’s tokenizer.

III. PRELIMINARIES

To establish the groundwork for subsequent sections, we introduce the **system log**, which records the system’s events and internal states during runtime. A system log contains a list of log messages in chronological order.

Fig. 1 presents a snippet of a raw system log generated by the BGL (the BlueGene/L supercomputer system), with each log message ordered according to the recorded time. These raw log messages are semi-structured texts consisting of a **header** and **content**. The header, determined by the logging framework, includes information such as timestamp, verbosity level (e.g., WARN/INFO), and component [47]. The log content comprises a constant part (keywords that reveal the log template) and a variable part (parameters that carry dynamic runtime information). In this paper, we focus solely on the content of each log message.

The log messages can be grouped into **log sequences** (i.e., series of log messages that record specific execution flows) based on session or fixed/sliding windows [48]. **Session window** partitioning groups log messages according to their session IDs, thereby generating sequences that include the log

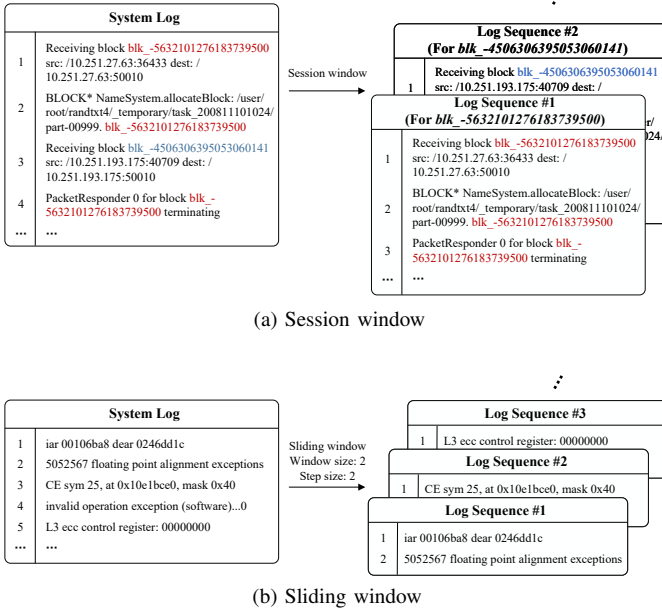


Fig. 2: Illustrative examples of log message partitioning.

messages within each session. For example, Fig. 2a illustrates the HDFS [49] logs undergoing the session window grouping process, where the *block_id* serves as the session ID. In contrast, **fixed/sliding window** partitioning groups log messages based on a fixed size (window size), which can be defined by either the time span or the number of log messages. This method creates sequences that capture snapshots of system log messages over time. For example, Fig. 2b illustrates the BGL [50] logs undergoing the sliding window grouping process, with a window size of 2 messages and a step size of 2 messages.

The objective of log-based anomaly detection is to identify anomalous log sequences, facilitating the recognition of potential issues within the system's operational behavior.

IV. METHODOLOGY

In this section, we present our innovative anomaly detection framework, LogLLM. As illustrated in Fig. 3, the log sequence undergoes preprocessing using regular expressions before being fed into a deep neural network that integrates BERT [43], a projector, and Llama [26] for log sequence classification. In the following sections, we will provide detailed insights into log sequence preprocessing, the architecture of the deep model, and the model training procedure.

A. Preprocessing

Considering that the log message content includes variable parameters carrying dynamic runtime information, which is always irrelevant to the anomalies and complicates deep model training, as demonstrated in Section V-F, a technique is needed to identify these parameters and replace them with a constant token. Log parsers, such as Drain [51] and Spell [52], are widely adopted in log-based anomaly detection methods and appear to be a useful technique. However, as noted by Le et al.

[3], existing log parsers do not always perform correctly on all log datasets and struggle to handle out-of-vocabulary (OOV) words in new log messages, resulting in a loss of semantic information. When logs are unstable, these parsers become increasingly ineffective over time, making it difficult to support subsequent anomaly detection.

Thanks to the structured log generation process, the textual format of parameters representing specific objects can be easily identified using regular expressions [53]. Consequently, we replace each variable parameter, such as account, directory path, and IP address, with '<*>'. Despite its simplicity, this technique offers significant performance advantages. Compared with log parsers, this preprocessing technique is more effective and does not require training.

B. Model Architecture

As shown in Fig. 3, our deep model consists of three main components: BERT, a projector, and Llama. Both BERT and Llama are pretrained LLMs. BERT is utilized to extract vector representations of log messages, while Llama is employed to classify the log sequences. The projector serves as a bridge, aligning the vector representation spaces of BERT and Llama. It is important to note that our model incorporates only one instance of BERT and one projector.

1) *BERT*: BERT generates a semantic vector by processing the semantic vector of the classification token ([CLS]) through a linear layer followed by a *tanh* activation function. Each log message, once preprocessed, is encoded into a semantic vector using the BERT tokenizer and BERT model. For a preprocessed log sequence, the output of BERT is a sequence of semantic vectors $C = (c_1, c_2, \dots, c_N) \in \mathbb{R}^{N \times d_{BERT}}$, where N represents the length of the log sequence (i.e., the number of log messages) and d_{BERT} is the dimension of each semantic vector (i.e., hidden size).

2) *Projector*: The projector is a linear layer that maps the semantic vectors $C \in \mathbb{R}^{N \times d_{BERT}}$ to the token embedding vectors accepted by Llama, represented as $E = (e_1, e_2, \dots, e_N) \in \mathbb{R}^{N \times d_{Llama}}$, where d_{Llama} is the hidden size of Llama. The projector is designed to align the vector representation spaces of BERT and Llama.

3) *Llama*: To conduct prompt tuning on Llama, the transformer decoder-based LLM, we generate corresponding textual queries based on embedded log sequences. Specifically, each query consists of three components.

The first component introduces the log sequence, such as "Below is a sequence of system log messages:". The second component comprises the token embeddings E output by the projector. The third component queries whether the sequence is anomalous, asking, for instance, ". Is this sequence normal or anomalous?". The first and third components are fed into the Llama tokenizer and Llama embedding layer sequentially, producing $E_1 \in \mathbb{R}^{A \times d_{Llama}}$ and $E_3 \in \mathbb{R}^{Q \times d_{Llama}}$, where A and Q are the number of tokens produced by tokenizing the first and third components, respectively. Then, the token embeddings of the three components are concatenated, rep-

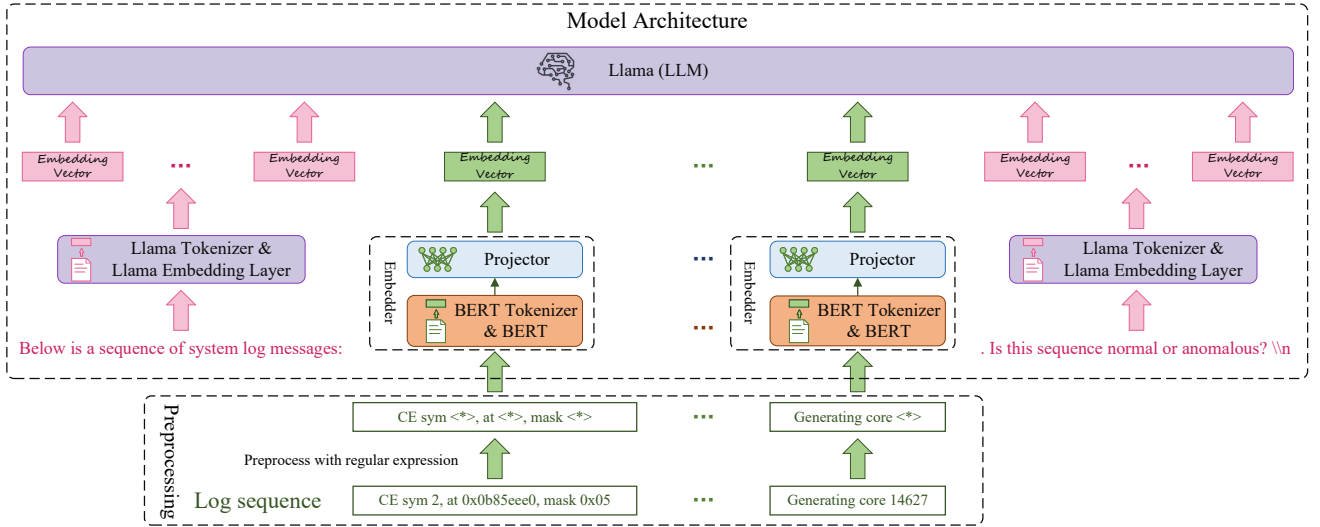


Fig. 3: The framework of LogLLM. Notably, the model includes a single instance of BERT and the projector.

resented as $[E_1||E||E_3] \in \mathbb{R}^{(A+N+Q) \times d_{Llama}}$ and fed into Llama.

C. Training

1) *Minority Class Oversampling*: LogLLM is a supervised anomaly detection method, which means it needs labeled normal and anomalous samples for training. However, supervised anomaly detection methods often face the challenge of data imbalance, which can lead to biased model training. In an anomaly detection task, there are only two classes: normal and anomalous, and the number of instances in each class is uncertain. To cope with data imbalance, we oversample the class with fewer samples, ensuring that the proportion of the minority class is no less than β . Formally, let the proportion of the minority class be α and $\alpha < \beta$, and the total number of samples be $Sample_num$. To achieve a proportion of β for the minority class, it will be oversampled to the following quantity:

$$\frac{\beta(1-\alpha)}{1-\beta} \times Sample_num \quad (1)$$

This adjustment will make the proportion of the minority class equal to β .

2) *Training Objective*: Our objective is to train the deep model to predict whether a given log sequence is normal or anomalous. We fine-tune the model to respond appropriately: if the sequence is anomalous, it outputs ‘The sequence is anomalous.’; if normal, it outputs ‘The sequence is normal.’. We utilize cross-entropy loss [54] as our loss function.

3) *Training Procedure*: To train our deep model, we follow three main stages.

Stage 1. Fine-tuning Llama to capture the answer template: The first stage involves fine-tuning Llama to capture the answer template. Specifically, we train Llama to respond to the prompt ‘Is this sequence normal or anomalous?’ with ‘The sequence is anomalous/normal.’. This stage requires only a few data samples.

Stage 2. Training the embedder of log messages: The second stage involves training the embedder of log messages, specifically BERT and the projector. This stage aims to project each log message to the embedding of the most suitable token in Llama, enabling Llama to discern whether the given log sequence is normal or anomalous.

Stage 3. Fine-tuning the entire model: Finally, we fine-tune the entire model to ensure cohesive and accurate performance across all components.

4) *Efficient Fine-Tuning on LLMs*: To reduce the costs involved in fine-tuning LLMs (BERT and Llama) with a substantial number of parameters, we utilize QLoRA [55] to minimize memory usage. QLoRA accomplishes this by backpropagating gradients into a frozen 4-bit quantized model, while maintaining the performance levels achieved during the full 16-bit fine-tuning process.

V. EXPERIMENTS

In this section, we conduct extensive experiments on four real-life logs to investigate the following research questions (RQs):

- **RQ1**: How effective is LogLLM in log-based anomaly detection?
- **RQ2**: How do different preprocessing techniques impact the performance of LogLLM?
- **RQ3**: How effective is the embedder for Llama?
- **RQ4**: How does the size of the Llama model affect the performance of LogLLM?
- **RQ5**: How does each stage of the three-stage training process influence the performance of LogLLM?
- **RQ6**: How do different levels of minority class oversampling, determined by the hyperparameter β , affect the performance of LogLLM?

LogLLM is coded in Python, and the source code is available at <https://github.com/guanwei49/LogLLM>.

A. Benchmark Methods

To verify the superiority of the proposed method, we compare LogLLM with five state-of-the-art semi-supervised methods: DeepLog [8], LogAnomaly [9], PLELog [22], FastLogAD [34], and LogBERT [32]. We also compare it with three supervised methods: LogRobust [19], CNN [18] and NeuralLog [3], and one method that does not require training a deep model but needs some normal samples for retrieval: RAPID [39].

Notably, FastLogAD, LogBERT, NeuralLog, and RAPID adopt LLMs for anomaly detection.

B. Experimental Settings

We conduct all experiments on a server equipped with an Intel Xeon Gold 6330 CPU (38 cores), 256GB of memory, and an NVIDIA A40 GPU with 48 GB of memory.

In our experiment, we utilize the BERT-base model¹ and Llama-3-8B model² as backbones. The hyperparameter β , which is described in Section IV-C1, is set to 30%. We use the AdamW optimizer [56] to train the model with a mini-batch size of 16. Unless otherwise specified, the training procedure is configured as follows: In the first stage, only 1,000 samples are involved with a learning rate of $5e-4$. The second and third stages each consist of two epochs with a learning rate of $5e-5$.

For a fair comparison, we configure the hyperparameters for all compared methods according to the values provided in their original articles.

C. Metrics

We evaluate the performance of these methods using the widely adopted *Precision*, *Recall* and F_1 - *score*. These metrics are calculated as follows:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F_1\text{-score} = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4)$$

, where TP , FN , FP represent true positives, false negatives and false positives respectively.

Precision refers to the percentage of correctly detected anomalies among all anomalies identified by the model, while recall represents the percentage of anomalies that are correctly identified from all real anomalies. The F_1 -score combines these two metrics into a single measure, providing a balanced assessment of the model's performance in detecting anomalies.

D. Dataset

To evaluate our method for log-based anomaly detection, we selected four public datasets [57]: HDFS, BGL, Liberty, and Thunderbird. The details for each dataset are provided below:

HDFS (Hadoop Distributed File System) dataset [49] is generated by running Hadoop-based mapreduce jobs on over

200 Amazon EC2 nodes and contains a total of 11,175,629 log messages. These log messages are grouped into different log windows based on their *block_id*, which reflect program executions in the HDFS, resulting in 575,061 blocks. Among these, 16,838 blocks (2.93%) indicate system anomalies.

BGL (Blue Gene/L) dataset [50] is a supercomputing system log dataset collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL). The dataset contains 4,747,963 log messages, each of which has been manually labeled as either normal or anomalous. There are 348,460 log messages (7.34%) that are labeled as anomalous.

Thunderbird dataset [50] is a publicly accessible collection of log data sourced from the Thunderbird supercomputer at Sandia National Laboratories (SNL). This dataset consists of both normal and anomalous messages, each of which has been manually categorized. Although the dataset contains over 200 million log messages, we focus on a subset of 10 million continuous log messages for computational efficiency. This subset includes 4,937 anomalous log messages, representing approximately 0.049% of the total.

Liberty dataset [50] comprises system logs from the Liberty supercomputer at Sandia National Labs (SNL) in Albuquerque. This supercomputer features 512 processors and 944 GB of memory, and the dataset contains over 200 million log messages. For computational efficiency, we sample 5 million consecutive log messages, among which 1,600,525 are identified as anomalous, constituting approximately 32.01% of the total sampled messages.

In the context of HDFS, we adopt a session window strategy, which involves grouping log messages into sequences based on the *block_id* present in each log message. Each session is labeled using ground truth. For other datasets, including BGL, Thunderbird, and Liberty, we utilize a sliding window strategy to group log messages, with a window size of 100 messages and a step size of 100 messages. A log sequence is deemed anomalous if it contains at least one anomalous log message according to the ground truth.

Similar to existing work [8], [9], [19], [22], [34], [39], we split each dataset into a training set and a testing set with a ratio of 8:2 to evaluate the performance of a log-based anomaly detection approach. For the HDFS dataset, we randomly split the log sequences into training and testing data. In contrast, for the BGL, Thunderbird, and Liberty datasets, we adhere to a chronological split [6]. This strategy ensures that all log sequences in the training set precede those in the testing set, reflecting real-world conditions and mitigating potential data leakage from unstable log data.

Table I summarizes the statistics of the datasets used in the experiments.

E. Performance Evaluation (RQ1)

Table II presents the experimental results of various log-based anomaly detection methods on the HDFS, BGL, Liberty, and Thunderbird datasets. The best results are highlighted in bold. We have the following observations:

¹<https://huggingface.co/google-bert/bert-base-uncased>

²<https://huggingface.co/meta-llama/Meta-Llama-3-8B>

TABLE I: The statistics of datasets used in the experiments.

	# Log messages	# Log sequences	Training Data			Testing Data		
			# Log sequences	# Anomalies	Anomaly ratio	# Log sequences	# Anomalies	Anomaly ratio
HDFS	11,175,629	575,061	460,048	13,497	2.93%	115,013	3,341	2.90%
BGL	4,747,963	47,135	37,708	4,009	10.63%	9,427	817	8.67%
Liberty	5,000,000	50,000	40,000	34,144	85.36%	10,000	651	6.51%
Thunderbird	10,000,000	99,997	79,997	837	1.05%	20,000	29	0.15%

TABLE II: Experimental results on HDFS, BGL, Liberty, and Thunderbird datasets. The best results are highlighted in bold.

Methods	Datasets	Log parser	HDFS			BGL			Liberty			Thunderbird			Avg. F ₁
			Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	
DeepLog		✓	0.835	0.994	0.908	0.166	0.988	0.285	0.751	0.855	0.800	0.017	0.966	0.033	0.506
LogAnomaly		✓	0.886	0.893	0.966	0.176	0.985	0.299	0.684	0.876	0.768	0.025	0.966	0.050	0.521
PLELog		✓	0.893	0.979	0.934	0.595	0.880	0.710	0.795	0.874	0.832	0.808	0.724	0.764	0.810
FastLogAD		✓	0.721	0.893	0.798	0.167	1.000	0.287	0.151	0.999	0.263	0.008	0.931	0.017	0.341
LogBERT		✓	0.989	0.614	0.758	0.165	0.989	0.283	0.902	0.633	0.744	0.022	0.172	0.039	0.456
LogRobust		✓	0.961	1.000	0.980	0.696	0.968	0.810	0.695	0.979	0.813	0.318	1.000	0.482	0.771
CNN		✓	0.966	1.000	0.982	0.698	0.965	0.810	0.580	0.914	0.709	0.870	0.690	0.769	0.818
NeuralLog		✗	0.971	0.988	0.979	0.792	0.884	0.835	0.875	0.926	0.900	0.794	0.931	0.857	0.893
RAPID		✗	1.000	0.859	0.924	0.874	0.399	0.548	0.911	0.611	0.732	0.200	0.207	0.203	0.602
LogLLM		✗	0.994	1.000	0.997	0.861	0.979	0.916	0.992	0.926	0.958	0.966	0.966	0.966	0.959

The proposed LogLLM achieves the highest F₁-score across all datasets. On average, LogLLM’s F₁-scores are 6.6% better than the best existing method, NeuralLog, demonstrating its effectiveness in log-based anomaly detection. Despite the adoption of LLMs in FastLogAD, LogBERT, NeuralLog, and RAPID for anomaly detection, their performance remains unsatisfactory. FastLogAD and LogBERT utilize BERT, a transformer encoder-based model, for detecting anomalies based on log sequence reconstruction errors. Their inputs consist of sequences of log template IDs (IDs of log string templates) extracted from log messages via log parsers, lacking semantic information. In contrast, NeuralLog and RAPID utilize transformer encoder-based models to extract semantic vectors from log messages. However, NeuralLog utilizes smaller models, whereas RAPID relies on distance-based comparison for anomaly sequence classification. LogLLM, on the other hand, leverages both BERT for extracting semantic vectors and Llama, a transformer decoder-based LLM, for anomaly detection. The representation spaces of BERT and Llama are aligned via a projector, fully harnessing the potential of LLMs for log-based anomaly detection.

Moreover, LogLLM achieves a balance between precision and recall, indicating that it maintains low false alarm rates and minimizes missed reports. In contrast, methods like FastLogAD are excessively sensitive to anomalies, often resulting in numerous false alarms. For example, on the BGL dataset, despite FastLogAD having a recall of 1, it only achieves a precision of 0.167, making it impractical for real-world use. Similarly, methods such as DeepLog, LogAnomaly and LogBERT exhibit similar issues. On the other hand, RAPID is not sensitive enough to anomalies, leading to many undetected anomalies. For instance, on the BGL dataset, RAPID achieves a precision of 0.874 but a recall of only 0.399.

Effect of labeled anomalies: As illustrated in Table II, in

TABLE III: Computational cost.

	Training time (Minutes)	Testing time (Minutes)
DeepLog	72.17	3.42
LogAnomaly	156.16	7.25
PLELog	315.47	33.59
LogRobust	108.42	2.48
CNN	98.16	2.16
FastLogAD	254.17	0.29
LogBERT	429.04	43.77
NeuralLog	267.46	21.44
RAPID	63.98	38.43
LogLLM	1,065.15	64.48

contrast to methods such as DeepLog, LogAnomaly, FastLogAD, LogBERT, and RAPID, which require clean datasets devoid of anomalies to build anomaly detection models, methods like PLELog, LogRobust, CNN, NeuralLog, and LogLLM demonstrate superior performance. These models are trained using not only normal samples but also labeled anomalies. For instance, these five methods achieve an average F₁-score above 0.771 across four datasets, whereas others that do not utilize labeled anomalies perform poorly, with an average F₁-score below 0.602 across four datasets. This demonstrates that incorporating labeled anomalies can provide a significant advantage to anomaly detection methods.

Computational cost: The time consumption of each method is presented in Table III. These results have been averaged across all the datasets.

Although RAPID does not require training a deep model, the extraction and retrieval of vector representations remain time-consuming. In comparison to other methods, FastLogAD requires relatively high training time, but it has the shortest testing time because it uses only the discriminator of the model during testing. As anticipated, while our proposed LogLLM demonstrates the best performance, it also incurs the highest

TABLE IV: Effects of different preprocessing techniques on HDFS, BGL, Liberty, and Thunderbird datasets. The best results are highlighted in bold.

	HDFS			BGL			Liberty			Thunderbird			Avg. F ₁
	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	
Raw	0.994	0.991	0.993	0.943	0.767	0.846	0.911	0.908	0.909	0.806	0.862	0.833	0.895
Template ID	0.995	0.945	0.969	0.775	0.286	0.418	0.994	0.270	0.425	1.000	0.379	0.550	0.591
Template	0.991	1.000	0.995	0.861	0.919	0.889	0.968	0.931	0.949	0.950	0.655	0.776	0.902
RE (LogLLM)	0.994	1.000	0.997	0.861	0.979	0.916	0.992	0.926	0.958	0.966	0.966	0.966	0.959

TABLE V: Effects of the embedder (BERT & adapter) and LLaMA model size, where ‘Mem.’ indicates GPU memory usage (GB), and ‘Tim.’ indicates training time (Minutes). ‘-’ indicates an out-of-memory (OOM) error.

	HDFS					BGL					Liberty					Thunderbird				
	Prec.	Rec.	F ₁	Mem.	Tim.	Prec.	Rec.	F ₁	Mem.	Tim.	Prec.	Rec.	F ₁	Mem.	Tim.	Prec.	Rec.	F ₁	Mem.	Tim.
L.-1B	0.986	0.995	0.991	16.5	1022.1	-	-	-	-	-	0.960	0.699	0.809	42.6	443.2	1.000	0.724	0.840	44.5	1732.1
Emb. & L.-1B	0.996	0.996	0.996	8.0	1412.2	0.734	0.944	0.825	32.4	187.1	0.950	0.905	0.927	29.3	173.2	0.875	0.966	0.918	32.4	715.1
L.-8B	0.988	0.997	0.992	43.0	4712.1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Emb. & L.-8B	0.994	1.000	0.997	16.6	2168.2	0.861	0.979	0.916	38.0	396.2	0.992	0.926	0.958	36.1	412.1	0.966	0.966	0.966	38.2	1284.2

computational cost due to its large number of parameters. However, the testing time of LogLLM remains acceptable when compared to other methods that utilize LLMs, such as LogBERT, NeuralLog, and RAPID.

F. Different Preprocessing Techniques (RQ2)

We evaluate the effectiveness of the different preprocessing techniques. The results are shown in Table IV. In this table, ‘Raw’ indicates that the content of log messages is not preprocessed and is directly input into the proposed deep model. ‘Template’ indicates that sequences of log templates produced by Drain [51], a log parser, are used as input for the proposed deep model. ‘Template ID’ signifies that the IDs of log templates, obtained by Drain, are simply encoded into numeric vectors using an embedding layer instead of BERT. The preprocessing technique ‘Template ID’ renders the model unable to capture the semantic information within log messages. Notably, the parser Drain is applied to the entire dataset, rather than only the training dataset, to avoid performance degradation due to the OOV problem. ‘RE’ indicates that regular expressions, as introduced in Section IV-A, are used for preprocessing log messages.

As anticipated, the preprocessing technique ‘RE’ yields the highest F₁-score across all datasets. Conversely, the preprocessing technique ‘Template ID’ consistently results in the lowest F₁-score across all datasets, averaging 36.8% lower than that of ‘RE’. This can be attributed to the fact that ‘Template ID’ hinders the model’s ability to capture the semantic information within log messages, thereby impairing its capability to detect anomalies from a natural language perspective. The preprocessing techniques ‘Raw’ and ‘Template’ result in relatively good performance, but their F₁-scores are still 6.4% and 5.7% lower than that of ‘RE’, respectively. For the preprocessing technique ‘Raw’, the variable parts (parameters that carry dynamic runtime information) within the content of each log message have little influence on anomaly detection. However, due to their high randomness, they can confuse

the model, making it difficult to discern anomalies. For the preprocessing technique ‘Template’, the parser is not always reliable, sometimes incorrectly removing the constant parts or retaining the variable parts, which can lead to information loss or confusion for the model, making it difficult to discern anomalies.

G. Effect of the Embedder (RQ3)

We investigate whether the embedder (BERT and adapter) is necessary for LogLLM. The results are presented in Table V. ‘L.-1B’ refers to directly inputting the log sequence (by concatenating log messages with semicolons (;) as separators into a long string) into the ‘Llama-3.2-1B’ model³. ‘Emb. & L.-1B’ represents LogLLM based on ‘Llama-3.2-1B’.

As expected, with the assistance of the embedder, the model requires less GPU memory, thereby avoiding out-of-memory (OOM) errors. Additionally, it enhances model performance by clarifying the boundaries between messages within a sequence. This improved representation enables the LLM to capture sequential dependencies better.

H. Effect of the Llama Model Size (RQ4)

As shown in Table V, larger LLaMA model sizes lead to better performance, at the cost of increased GPU memory usage and longer training times.

On average, compared to using Llama-3.2-1B, adopting Llama-3-8B improves the F₁-score by 4.3%, but increases GPU memory usage by 7.7 GB and extends training time by 443.2 minutes.

I. Ablation Study of the Training Procedure (RQ5)

We investigate the effect of each training procedure through an ablation study. The results are presented in Table VI, where ‘W/O’ denotes ‘without’. We have the following observations:

Skipping any training stage results in a decrease in the F₁-score across all datasets, demonstrating the effectiveness of our

³<https://huggingface.co/meta-llama/Llama-3.2-1B>

TABLE VI: Ablation study of the training procedure on HDFS, BGL, Liberty, and Thunderbird datasets. The best results are highlighted in bold.

	HDFS			BGL			Liberty			Thunderbird			Avg. F ₁
	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	Prec.	Rec.	F ₁	
W/O Stage 1	0.991	1.000	0.995	0.578	0.971	0.725	0.685	0.290	0.408	0.381	0.828	0.522	0.662
W/O Stage 2	0.994	1.000	0.997	0.858	0.920	0.888	0.995	0.906	0.949	0.848	0.966	0.903	0.934
W/O Stage 1&2	0.992	1.000	0.996	0.853	0.882	0.868	0.995	0.906	0.949	0.897	0.897	0.897	0.927
W/O Stage 3	0.993	0.999	0.996	0.704	0.776	0.738	1.000	0.684	0.812	0.958	0.793	0.868	0.854
LogLLM	0.994	1.000	0.997	0.861	0.979	0.916	0.992	0.926	0.958	0.966	0.966	0.966	0.959

three-stage training procedure. It is noteworthy that training without stage 1 leads to the worst performance, with the F₁-score averaged across all datasets decreasing by as much as 29.7%. However, training without stages 1&2 (only adopting training stage 3: fine-tuning the entire model) yields acceptable performance, with only a 3.2% decrease in the average F₁-score. This demonstrates that fine-tuning Llama to capture the answer template (Stage 1) is essential before training the embedder (BERT and projector) of log messages (Stage 2). Without stage 1 (i.e., directly training the embedder), the embedder may be misdirected, resulting in incorrect semantic capture of log messages and model failure. Training without stage 3 yields relatively poor performance, with an average F₁-score decrease of 10.5%. This indicates that sequentially fine-tuning Llama and training the embedder alone is insufficient for the model to capture anomalous patterns; cohesive fine-tuning of the entire model is essential. Training without stages 2 and 1&2 also results in a performance decrease, with average F₁-score reductions of 2.5% and 3.2%, respectively. This demonstrates that individually training the embedder before fine-tuning the entire model can also enhance performance. This stage allows the embedder to generate better semantic vectors of log messages for Llama to discern anomalies.

In summary, our proposed three-stage training procedure is well-suited for our deep model in log-based anomaly detection.

J. Impact of Minority Class Oversampling (RQ6)

Note that normal and anomalous samples in the training dataset are imbalanced, as shown in Table I. For the HDFS, BGL, and Thunderbird datasets, normal samples outnumber anomalous samples. Conversely, in the Liberty dataset, anomalous samples exceed normal samples. As described in Section IV-C1, the hyper-parameter β controls the proportion of the minority class by oversampling to address the data imbalance problem. In this section, we investigate the impact of β by varying its value. Fig. 4 illustrates the performance of LogLLM on the four datasets under different magnitudes of β . When $\beta = 0$, the samples are not oversampled; instead, the original datasets are utilized directly for training.

As illustrated in Fig. 4b, for the HDFS, BGL, and Thunderbird datasets, the recall always increases, while for the Liberty dataset, recall decreases as β increases. This can be attributed to the fact that for the HDFS, BGL, and Thunderbird datasets, when β increases, anomalies are oversampled, making the model more prone to identifying samples as anomalies. In

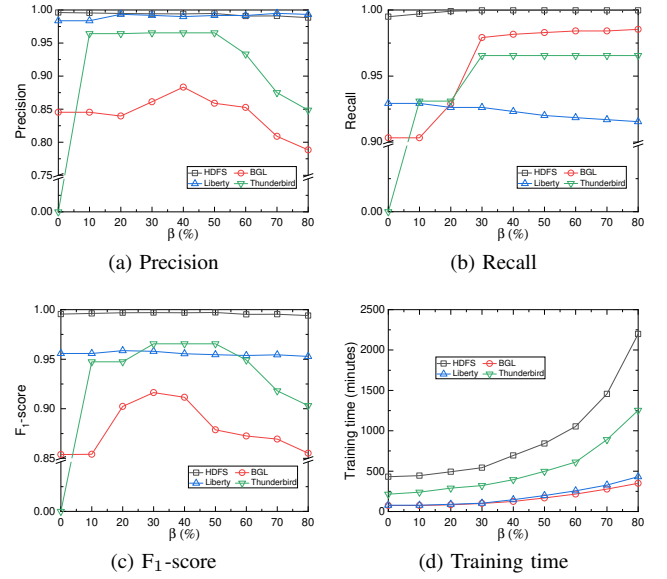


Fig. 4: Impact of minority class oversampling.

contrast, for the Liberty dataset, when β increases, normal samples are oversampled, making the model more prone to identifying samples as normal.

As illustrated in Fig. 4c, the trend of the F₁-score is basically the same across all datasets. The F₁-score increases and then decreases as β increases. However, the LogLLM seems not to be sensitive to β ; when β is between 10% and 80%, the variation in the F₁-score is no more than 0.07. Thanks to the substantial semantic knowledge embedded in LLMs, a trained model can effectively learn anomalous patterns and detect anomalies, even when the minority class constitutes only 10% of the dataset. However, LogLLM appears unable to effectively handle extremely imbalanced scenarios. For instance, in the Thunderbird dataset, anomalies constitute only 1.05% of the samples, causing the trained model to be biased and classify all samples as normal. As a result, precision, recall, and F₁-score are all equal to 0.

Compared to the BGL and Thunderbird datasets, the precision, recall and F₁-score for the HDFS and Liberty datasets exhibit minimal variation with respect to β . This consistency arises from the more distinct patterns between abnormal and normal samples in the HDFS and Liberty datasets, allowing LogLLM to easily differentiate them, regardless of the ratio

of normal and abnormal samples.

As anticipated, as β increases, the training time also increases, as shown in Fig. 4d. This relationship arises because a higher β leads to more oversampled data samples, as indicated by equation (1), thereby enlarging the training dataset.

To summarize, minority class oversampling is essential; however, the value of the hyperparameter β does not significantly impact the performance of LogLLM, making careful selection unnecessary. Moreover, excessively large values of β are undesirable, as they result in prolonged training times. Values between 30% and 50% are deemed acceptable.

VI. CONCLUSION

In this paper, we propose LogLLM, a novel log-based anomaly detection framework that leverages LLMs. LogLLM employs both transformer encoder-based and decoder-based LLMs, specifically BERT and Llama, for log-based anomaly detection. BERT is utilized to extract semantic vectors from log messages, while Llama is used to classify log sequences. To ensure coherence in log semantics, we introduce a projector that aligns the vector representation spaces of BERT and Llama. LogLLM is trained using an innovative three-stage procedure designed to enhance both performance and adaptability. Extensive experiments conducted on four public real-world datasets demonstrate that LogLLM achieves remarkable performance. Subsequent ablation studies further confirm the effectiveness of our three-stage training procedure.

REFERENCES

- [1] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 41–50.
- [2] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.
- [3] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 492–504.
- [4] W. Guan, J. Cao, H. Zhao, Y. Gu, and S. Qian, "Survey and benchmark of anomaly detection in business processes," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–23, 2024.
- [5] S. Zhang, Y. Ji, J. Luan, X. Nie, Z. Chen, M. Ma, Y. Sun, and D. Pei, "End-to-end automl for unsupervised log anomaly detection," *Automated Software Engineering (ASE'24)*, 2024.
- [6] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1356–1367.
- [7] J. Qi, S. Huang, Z. Luan, S. Yang, C. Fung, H. Yang, D. Qian, J. Shang, Z. Xiao, and Z. Wu, "Loggpt: Exploring chatgpt for log-based anomaly detection," in *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2023, pp. 273–280.
- [8] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.
- [9] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *IJCAI*, vol. 19, no. 7, 2019, pp. 4739–4745.
- [10] L. Zhang, W. Li, Z. Zhang, Q. Lu, C. Hou, P. Hu, T. Gui, and S. Lu, "Logattn: Unsupervised log anomaly detection with an autoencoder based attention mechanism," in *International conference on knowledge science, engineering and management*. Springer, 2021, pp. 222–235.
- [11] M. Catillo, A. Pecchia, and U. Villano, "Autolog: Anomaly detection by deep autoencoding of system logs," *Expert Systems with Applications*, vol. 191, p. 116263, 2022.
- [12] Y. Xie and K. Yang, "Log anomaly detection by adversarial autoencoders with graph feature fusion," *IEEE Transactions on Reliability*, 2023.
- [13] X. Zhang, X. Chai, M. Yu, and D. Qiu, "Anomaly detection model for log based on lstm network and variational autoencoder," in *2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*. IEEE, 2023, pp. 239–244.
- [14] X. Duan, S. Ying, W. Yuan, H. Cheng, and X. Yin, "A generative adversarial networks for log anomaly detection," *Comput. Syst. Sci. Eng.*, vol. 37, no. 1, pp. 135–148, 2021.
- [15] Z. He, Y. Tang, K. Zhao, J. Liu, and W. Chen, "Graph-based log anomaly detection via adversarial training," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2023, pp. 55–71.
- [16] C. Zhang, X. Wang, H. Zhang, J. Zhang, H. Zhang, C. Liu, and P. Han, "Layerlog: Log sequence anomaly detection based on hierarchical semantics," *Applied Soft Computing*, vol. 132, p. 109860, 2023.
- [17] S. Hashemi and M. Mäntylä, "Onelog: towards end-to-end software log anomaly detection," *Automated Software Engineering*, vol. 31, no. 2, p. 37, 2024.
- [18] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 151–158.
- [19] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 807–817.
- [20] Y. Xie, H. Zhang, and M. A. Babar, "Loggd: Detecting anomalies from system logs with graph neural networks," in *2022 IEEE 22nd International conference on software quality, reliability and security (QRS)*. IEEE, 2022, pp. 299–310.
- [21] Z. Zhao, W. Niu, X. Zhang, R. Zhang, Z. Yu, and C. Huang, "Trine: Syslog anomaly detection with three transformer encoders in one generative adversarial network," *Applied Intelligence*, vol. 52, no. 8, pp. 8810–8819, 2022.
- [22] 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.
- [23] S. Hochreiter, "Long short-term memory," *Neural Computation MIT Press*, 1997.
- [24] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.
- [25] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.
- [26] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
- [27] T. GLM, A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Rojas, G. Feng, H. Zhao, H. Lai *et al.*, "Chatglm: A family of large language models from glm-130b to glm-4 all tools," *arXiv preprint arXiv:2406.12793*, 2024.
- [28] W. Guan, J. Cao, J. Gao, H. Zhao, and S. Qian, "Dabl: Detecting semantic anomalies in business processes using large language models," *arXiv preprint arXiv:2406.15781*, 2024.
- [29] Y. Liu, S. Tao, W. Meng, F. Yao, X. Zhao, and H. Yang, "Logprompt: Prompt engineering towards zero-shot and interpretable log analysis," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 364–365.
- [30] C. Egersdoerfer, D. Zhang, and D. Dai, "Early exploration of using chatgpt for log-based anomaly detection on parallel file systems logs," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 315–316.
- [31] J. Pan, W. S. Liang, and Y. Yidi, "Raglog: Log anomaly detection using retrieval augmented generation," in *2024 IEEE World Forum on Public Safety Technology (WFPST)*. IEEE, 2024, pp. 169–174.

- [32] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," in *2021 international joint conference on neural networks (IJCNN)*. IEEE, 2021, pp. 1–8.
- [33] Y. Lee, J. Kim, and P. Kang, "Lanobert: System log anomaly detection based on bert masked language model," *Applied Soft Computing*, vol. 146, p. 110689, 2023.
- [34] Y. Lin, H. Deng, and X. Li, "Fastlogad: Log anomaly detection with mask-guided pseudo anomaly generation and discrimination," *arXiv preprint arXiv:2404.08750*, 2024.
- [35] C. Almodovar, F. Sabrina, S. Karimi, and S. Azad, "Logfit: Log anomaly detection using fine-tuned language models," *IEEE Transactions on Network and Service Management*, 2024.
- [36] S. Chen and H. Liao, "Bert-log: Anomaly detection for system logs based on pre-trained language model," *Applied Artificial Intelligence*, vol. 36, no. 1, p. 2145642, 2022.
- [37] J. L. Adeba, D.-H. Kim, and J. Kwak, "Sarlog: Semantic-aware robust log anomaly detection via bert-augmented contrastive learning," *IEEE Internet of Things Journal*, 2024.
- [38] Y. Fu, K. Liang, and J. Xu, "Mlog: Mogrifier lstm-based log anomaly detection approach using semantic representation," *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3537–3549, 2023.
- [39] G. No, Y. Lee, H. Kang, and P. Kang, "Training-free retrieval-based log anomaly detection with pre-trained language model considering token-level information," *Engineering Applications of Artificial Intelligence*, vol. 133, p. 108613, 2024.
- [40] F. Hadadi, Q. Xu, D. Bianculli, and L. Briand, "Anomaly detection on unstable logs with gpt models," *arXiv preprint arXiv:2406.07467*, 2024.
- [41] M. Burtsev, M. Reeves, and A. Job, "The working limitations of large language models," *MIT Sloan Management Review*, vol. 65, no. 2, pp. 8–10, 2024.
- [42] A. Joulin, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.
- [43] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
- [44] Y. Liu, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
- [45] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, "Spanbert: Improving pre-training by representing and predicting spans," *Transactions of the association for computational linguistics*, vol. 8, pp. 64–77, 2020.
- [46] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [47] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 121–130.
- [48] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2016, pp. 654–661.
- [49] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 588–597.
- [50] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*. IEEE, 2007, pp. 575–584.
- [51] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.
- [52] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.
- [53] V.-H. Le and H. Zhang, "Log parsing with prompt-based few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2438–2449.
- [54] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Neurocomputing: Algorithms, architectures and applications*. Springer, 1990, pp. 227–236.
- [55] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [56] I. Loshchilov, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.
- [57] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 355–366.

LogFormer: A Pre-train and Tuning Pipeline for Log Anomaly Detection

Hongcheng Guo¹, Jian Yang¹, Jiaheng Liu¹, Jiaqi Bai¹, Boyang Wang¹, Zhoujun Li^{1*}, Tieqiao Zheng², Bo Zhang², Junran Peng¹, Qi Tian³

¹State Key Lab of Software Development Environment, Beihang University, Beijing, China

²Cloudwise Research, Beijing, China

³Huawei, Beijing, China

{hongchengguo,jiaya,liujiaheng,bjq,wangboyang,lizj}@buaa.edu.cn {steven.zheng,bowen.zhang}@cloudwise.com

Abstract

Log anomaly detection is a key component in the field of artificial intelligence for IT operations (AIOps). Considering log data of variant domains, retraining the whole network for unknown domains is inefficient in real industrial scenarios. However, previous deep models merely focused on extracting the semantics of log sequences in the same domain, leading to poor generalization on multi-domain logs. To alleviate this issue, we propose a unified Transformer-based framework for Log anomaly detection (LogFormer) to improve the generalization ability across different domains, where we establish a two-stage process including the pre-training and adapter-based tuning stage. Specifically, our model is first pre-trained on the source domain to obtain shared semantic knowledge of log data. Then, we transfer such knowledge to the target domain via shared parameters. Besides, the Log-Attention module is proposed to supplement the information ignored by the log-paring. The proposed method is evaluated on three public and one real-world datasets. Experimental results on multiple benchmarks demonstrate the effectiveness of our LogFormer with fewer trainable parameters and lower training costs.

Introduction

With the rapid development of large-scale IT systems, numerous companies have an increasing demand for high-quality cloud services. Anomaly detection (Breier and Branišová 2015) is critical to monitor data peculiarities for logs, which describe detailed system events at runtime and the intention of users in the large-scale services (Zhang et al. 2015). It is error-prone to detect anomalous logs from a local perspective. In this case, some automatic detection methods based on machine learning are proposed (Xu et al. 2010). Due to the development of IT services, the volume of log data has grown fast and traditional approaches are infeasible. Meanwhile, as log messages are half-structured and have their semantics, it is similar to natural language corpus. Therefore, many deep learning methods based on language models (Hochreiter and Schmidhuber 1997; Devlin et al. 2018) have been proposed on log anomaly detection task (Zhang et al. 2016; Du et al. 2017; Zhang et al. 2019; Meng et al. 2019; Guo et al. 2023a). However, these models adopt parser (Du and Li 2016; He

BGL:	Unusual End of Program
data storage interrupt	
rts: kernel terminated for reason 1004rts: bad message header: [...]	
Thunderbird:	
kernel: mptscsih: ioc0: attempting task abort! (sc=00000101bddee480)	
Red Storm:	
DMT 310 Command Aborted : SCSI cmd:2A LUN 2 DMT 310 T:299 a: [...]	
BGL:	Program Not Running
rts panic! - stopping execution	
Thunderbird:	
pbs mom: Bad file descriptor (9) in tm request, job [job] not running	
Spirit:	
kernel: GM: LANai is not running . Allowing port=0 open for debugging	
Liberty:	
kernel: GM: LANai is not running . Allowing port=0 open for debugging	

Figure 1: The same anomaly from multiple domains. The top part denotes the “Unusual End of Program” anomaly from three domains including BGL, Thunderbird, and Red Storm while the bottom part is the “Program Not Running” from four domains including BGL, Thunderbird, Spirit, and Liberty.

et al. 2017) to gain templates in logs before detection, which leads to the loss of semantics in raw log data.

Despite being different in morphology and syntax, logs of multiple domains usually share similar semantic space. For example, in Fig. 1, three sources (BGL, Thunderbird, Red Storm) have the same anomaly called **Unusual End of Program**. However, existing methods mostly focus on a single domain. When the components from a new domain are introduced, these methods lack the ability to accommodate such unseen logs. Besides, we need to consider the continuous iteration of log data when the system upgrades and it is costly to retrain different models for different datasets.

In this paper, we address the problems above via a two-stage solution called LogFormer. LogFormer is capable of preserving the shared semantic knowledge between different domains. Specifically, to avoid information loss due to parsing, Log-Attention module is to supplement the information ignored by the log-paring for better performance. Then in the first stage, we create a model based on the Log-Attention, which is pre-trained on the source domain to obtain common semantics of log sequences. Second, LogFormer uses a flexible component called Adapter to transfer knowledge from the source domain to the target domain.

*Corresponding author.

Generally, the contributions are as follows: (i) We propose LogFormer, an end-to-end Pre-train and Tuning pipeline to automatically detect log anomalies, which provides a new perspective via simple and effective pre-training and adapter-based tuning strategies for log anomaly detection. (ii) Log-Attention module is proposed to avoid the loss of semantics caused by log parsing. (iii) With only a few additional trainable parameters on the target domain, the training costs are reduced a lot based on the effective parameter-sharing strategy in LogFormer. (iv) LogFormer achieves state-of-the-art performance on three public benchmark datasets.

Related Work

Log Parsing Developers can create an unlimited number of variable names, abbreviations, or special technical terms, which are out of the scale of ordinary English words. If we conduct word splitting, the endless unseen log tokens would explode the vocabulary, which is called the out-of-vocabulary (OOV) problem. To handle this issue, log parsing is used to convert unstructured logs into structured event templates by **keeping keywords** (Jiang et al. 2008; Mankanju, Zincir-Heywood, and Milios 2009; He et al. 2017) and **removing extra parameters**, where the parameters usually denote special fields (e.g., /etc, /tmp), words (e.g., *, _), serial numbers (e.g., 0x10001) and so on. In Fig. 2, we use Drain (He et al. 2017) to extract the templates, and each log and the corresponding template are matched. Then, the log template sequence is fed into anomaly detection models.

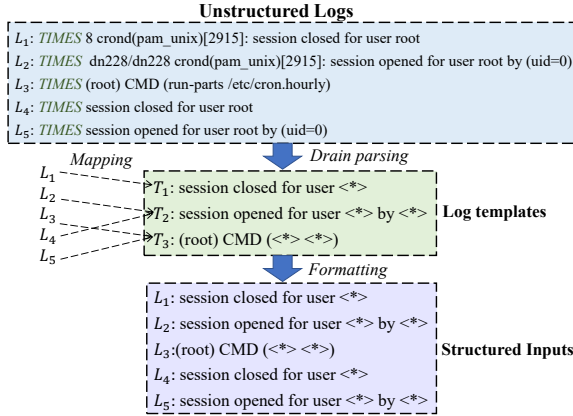


Figure 2: Logs and Templates. The top part is unstructured logs, we adopt Drain algorithm to extract log templates, then we match each log with its template, which is the middle part. The bottom part is structured inputs.

Log Anomaly Detection Natural language processing technology (Bai et al. 2023; Yang et al. 2022; Guo et al. 2022) is evolving rapidly, which has also reinvigorated this task. Supervised methods are based on classification. (Breier and Branišová 2015; Huang et al. 2020; Lu et al. 2018). LogRobust (Zhang et al. 2019) uses both normal and abnormal log data for training based on the Bi-LSTM architecture. Some semi-supervised methods (Xu et al. 2010; Yang et al. 2021b) are proposed to alleviate such burden. DeepLog (Du

et al. 2017) uses LSTM to forecast the next log sequence with ranked probabilities. Besides, LogAnomaly (Meng et al. 2019) uses log embeddings to capture the semantic information. PLElog (Yang et al. 2021a) clusters the features of normal data and detects the anomalies by GRU. Although these methods obtain performance improvements on existing log datasets from a single source, they ignore the shared semantics between multiple sources and the value of the parameters removed by log parsing.

Approach

In this section, we describe the general framework of LogFormer. In Fig. 3, LogFormer contains two stages: pre-training and adapter-based tuning. In the following, we present the definition of the problem, and the components of LogFormer. Finally, we introduce the processes of the two stages.

Problem Definition

Log anomaly detection problem is defined as a binary classification task. The model is supposed to determine whether the input log is abnormal or normal. For the source domain, after preprocessing each raw logs, we generate the vector representations of K_{src} log sequences, which are denoted as $S^{src} = \{S_k\}_{k=1}^{K_{src}}$. Then, $S_i^{src} = \{V_t^{src}\}_{t=1}^{T_i^{src}}$ denotes the i -th log sequence, where T_i^{src} is the length of the i -th log sequence and V_t^{src} denotes the t -th log sentence in S_i^{src} . For the target domain, $S^{tgt} = \{S_k^{tgt}\}_{k=1}^{K_{tgt}}$ denotes the representations of K_{tgt} log sequences. $S_j^{tgt} = \{V_t^{tgt}\}_{t=1}^{T_j^{tgt}}$ denotes the j -th log sequence, where T_j^{tgt} is the length of the j -th log sequence. Therefore, the training procedure is as follows. We first pre-train the model on the dataset from source domain as follows:

$$f_p(y_i | S_i^{src}; \Theta), \quad (1)$$

where f_p represents the pre-training stage and Θ is the model parameters in pre-training stage. Then, the model is transferred to the target domain:

$$f_a(y_j | S_j^{tgt}; \Theta_f, \theta_a). \quad (2)$$

where f_a represents the adapter-based tuning stage. Θ_f denotes the parameter of the encoder transferred from the pre-training stage, which is frozen in adapter-based tuning stage. θ_a is the parameter of the adapter. y is the ground-truth label.

Feature Extractor

The feature extractor converts session sequences (template sequence) to vectors with the same dimension d . Here we use the pre-trained sentence-bert (Reimers and Gurevych 2019) model to get the representation of the template sequence in Fig. 2. Suppose each session has l fixed length, then the embedding of the input X after the feature extractor:

$$X_E = \mathbf{FE}(X). \quad (3)$$

Where $\mathbf{FE}$ represents the encoder of sentence-bert. we can obtain $X_E \in \mathbb{R}^{l \times d}$ for each session.

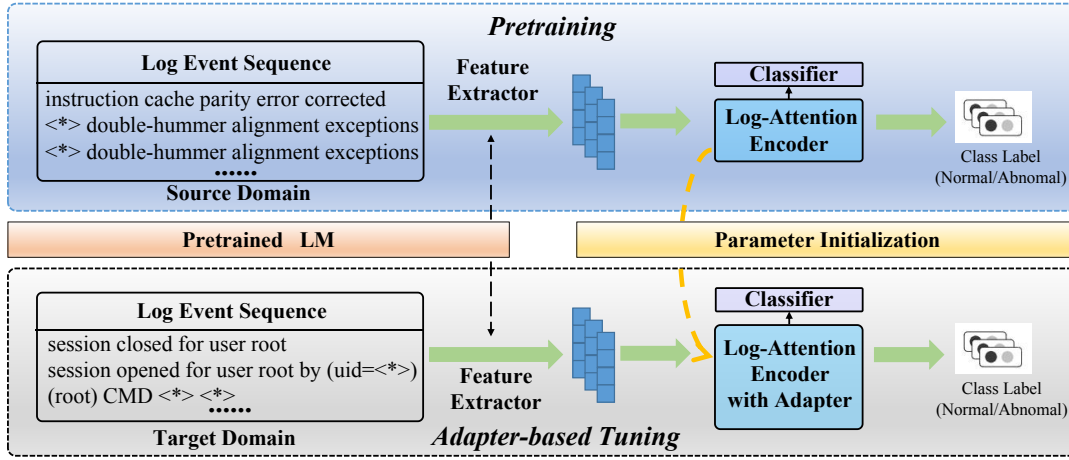


Figure 3: Overview of architecture. Log sequences are first fed into the pre-trained language model to extract features. The Log-Attention encoder is trained on the source domain to acquire shared semantic information. Then, we initialize the encoder and only tune the parameters of the adapter on the target domain to transfer the knowledge.

Log-Attention Module

Although parsing solves the out-of-vocabulary (OOV) problem, the information of the parameters is discarded. To aggregate parameters and keywords information, we have adjusted the structure of the original transformer encoder. Log-Attention module is proposed in Fig. 4. Specifically, after parsing, we gain the P parameters for each log sequence. For each character P_i in P , we adopt the feature extractor to obtain character-level embedding P_i^E . Then we use the Linear layer to encode the whole P^E as follows:

$$\phi_p = \text{LINEAR}(P^E). \quad (4)$$

where ϕ_p denotes the output of parameter encoding. Then, we assign each output a learnable scalar, which will serve as a bias term in self-attention. The intuition of Log-Attention is also inspired by position encoding, which is mapped as bias in attention and provides additional position information. The Log-Attention is computed as follows:

$$\text{LogAttention} = \text{Softmax}\left(\frac{QK^T}{\sqrt{d/h}} + \phi_p\right)V. \quad (5)$$

where h is the number of the heads, d denotes the dimension of the input, and Q, K, V represent queries, keys, and values, respectively.

Encoder with Adapter

The order of a log sequence conveys information about the program execution sequence. Wrong execution order is also considered abnormal. Thus, constant positional embedding is also used. The component after the attention layer and feed-forward layer is the original serial adapter. We design our log adapter with a parallel structure in Fig. 5, which is inserted parallel to the Log-Attention layer and feedforward layer. This design allows adapter to use input information better with original complete encoders. During adapter-based tuning, only a few parameters of the adapters are updated on the target domain. More specifically, we use down- and up-scale

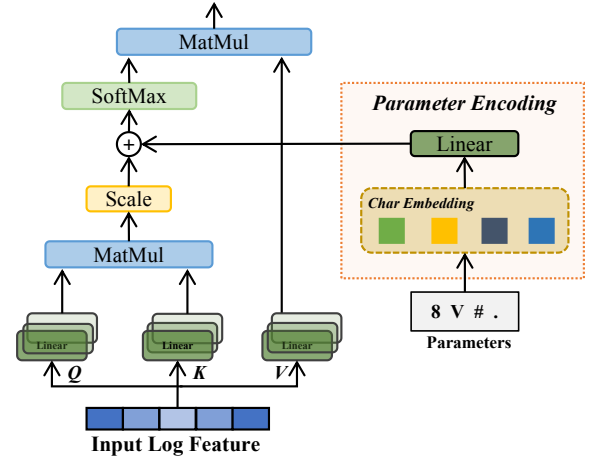


Figure 4: Log-Attention. The left part is the multi-head attention, and the right part is the parameter encoding.

neural networks as the adapter. Two projection layers first map the hidden vector from dimension d to dimension m and then map it back to d . The adapter also has a skip-connection operation internally. The output vector h' is calculated:

$$h' = W_{up} \tanh(W_{down}h) + h. \quad (6)$$

where $h \in \mathbb{R}^d$ represents a given hidden vector. $W_{down} \in \mathbb{R}^{m \times d}$ and $W_{up} \in \mathbb{R}^{d \times m}$ is the down-projection and the up-projection matrix respectively, by setting $m \ll d$, we limit the number of parameters added per adapter, which is the core to reduce trainable parameters while retaining semantic information.

Pre-training

Inspired by pre-trained models (Devlin et al. 2018; Reimers and Gurevych 2019; Guo et al. 2023b), we can acquire the common representation for log anomaly with the stacked log-attention encoders. In this stage, the pre-trained model learns

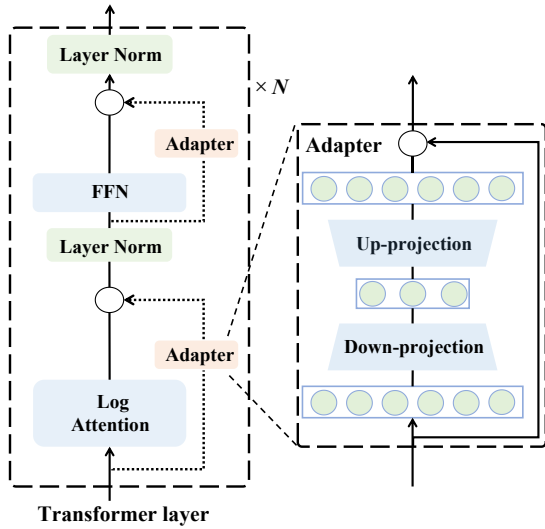


Figure 5: Encoder with Adapters. Where N is the number of encoder layers. The left part describes the log-attention encoder inserted by parallel adapters, and the right part is the structure of an adapter, which is composed of the down- and up-projection layers.

the commonalities among different anomalies. Specifically, the objective of pre-training is a supervised binary classification task without using log adapter. Then, the parameters of the log-attention encoders are used as the initialization of the next stage.

Adapter-based Tuning

Adapter-based tuning leverages the knowledge obtained from the pre-training with lightweight adapters (Houlsby et al. 2019). Specifically, in the second stage, we plug adapters into the encoders of the pre-trained model, where only the parameters of the adapters are updated during target domain adaption. Parameters of the Log-Attention and the feedforward layers in the pre-trained model are frozen. Unlike fine-tuning, LogFormer provides a plug-in mechanism to reuse the pre-trained model with only a few additional trainable parameters.

Training Strategy

The classifier is simply implemented by one linear layer. We both take BCE loss for two stages. Thus, the loss of the pre-training stage is as follows:

$$\mathcal{L}_p = -\mathbb{E}_{x,y \in D_{x,y}^{src}} [\log P(y|x; \Theta)], \quad (7)$$

where $\mathcal{L}_p$ represents the loss in the pre-training stage. Θ is the parameter of the whole model in the pre-training stage. x and y are the input data and label respectively, $D_{x,y}^{src}$ represents the data coming from the source domain. Then, we define the objective loss in the adapter-based tuning stage:

$$\mathcal{L}_a = -\mathbb{E}_{x,y \in D_{x,y}^{tgt}} [\log P(y|x; \Theta_f, \theta_a)]. \quad (8)$$

where $\mathcal{L}_a$ is the loss function in the adapter-based tuning stage. Θ_f is the parameter of the encoder module trained in

Dataset	HDFS	BGL	Thunderbird
Category	Distributed	Supercomputer	Supercomputer
#Messages	11M	5M	10M
#Anomaly	17K	40K	123K
#Templates	49	1423	1092
#Error Types	53	143	95

Table 1: A summary of the datasets used in this work. Messages are the raw log strings. Log sequences are extracted by ID or sliding window method.

the pre-training stage, which is frozen in the adapter-based tuning stage. θ_a is the parameter of the adapter. $D_{x,y}^{tgt}$ represents the data coming from the target domain.

Experiments

In this section, we compare our method with existing methods on multiple benchmark datasets.

Datasets We conduct experiments on three datasets from LogHub (He et al. 2020)¹. HDFS (Xu et al. 2010) dataset is generated and collected from the Amazon EC2 platform through running Hadoop-based map-reduce jobs. Thunderbird and BGL datasets (Oliner and Stearley 2007) contain logs collected from a two-supercomputer system at Sandia National Labs (SNL) in Albuquerque. The log contains alert and non-alert messages identified by alert category tags. Following (Yao et al. 2020; Meng et al. 2019), 10M/11M/5M continuous log messages from Thunderbird/HDFS/BGL are used.

Preprocessing We extract log sequences by **block IDs** in HDFS. For BGL and Thunderbird, we utilize the **sliding window** (size of 20) without overlap to generate log sequences. We adopt Drain (He et al. 2017) with specifically designed regex to do log parsing. For each dataset, considering that logs evolve over time, we select the first 80% (according to the timestamp of logs) log sequences for training and the rest 20% for testing, which is consistent with the prior work (Yang et al. 2021a; Du et al. 2017).

Implementation Details In experiments, we use different numbers of transformer encoder layers in $\{1, 2, 4\}$. The number of attention heads is 8, and the size of the feedforward network that takes the output of the multi-head self-attention mechanism is 3072. We use Adam as the optimization algorithm whose learning rate is scheduled by OneCycleLR, with $\beta_1 = 0.9$, $\beta_2 = 0.99$, and $\epsilon = 10^{-8}$. All runs are trained on 2 NVIDIA A100(40G) with a batch size of 64. For each dataset, we tune the maximum learning of the OneCycleLR scheduler in $\{1e-5, 5e-5, 1e-6\}$.

Baselines and Evaluation Table 1 shows five public baselines including Support Vector Machine(SVM), Deeplog (Du et al. 2017), LogAnomaly (Meng et al. 2019), LogRobust (Zhang et al. 2019), and PLELog (Yang et al. 2021a)² and

¹<https://github.com/logpai/loghub>

²<https://github.com/YangLin-George/PLELog>

Dataset	Method	Precision	Recall	F_1 Score
HDFS	SVM	0.31	0.65	0.41
	DeepLog	0.83	0.87	0.85
	LogAnomaly	0.86	0.89	0.87
	PLELog	0.88	0.93	0.90
	LogRobust	0.88	0.95	0.91
	ChatGPT	0.74	0.82	0.78
	LogFormer _S	0.95	0.96	0.95
	LogFormer _P	0.96	0.97	0.96
	LogFormer	0.97	0.98	0.98
BGL	SVM	0.22	0.56	0.32
	DeepLog	0.14	0.81	0.24
	LogAnomaly	0.19	0.78	0.31
	PLELog	0.92	0.96	0.94
	LogRobust	0.92	0.96	0.94
	ChatGPT	0.77	0.71	0.74
	LogFormer _S	0.96	0.97	0.97
Thunderbird	SVM	0.34	0.91	0.46
	DeepLog	0.48	0.89	0.62
	LogAnomaly	0.51	0.97	0.67
	PLELog	0.85	0.94	0.89
	LogRobust	0.89	0.96	0.92
	ChatGPT	0.84	0.79	0.81
	LogFormer _S	0.94	0.98	0.96
	LogFormer _P	0.97	0.99	0.98
	LogFormer	0.99	0.99	0.99

Table 2: Results on Thunderbird, BGL and HDFS. LogFormer_S represents the model trained from scratch, LogFormer_P represents the model trained with pre-training but tuning without adapters.

two variants of LogFormer³. We also compare with the popular ChatGPT (Ouyang et al. 2022). Here, LogFormer_S is trained from scratch without two stages. LogFormer_P is trained only with the pre-training stage, which means we directly tune the whole parameters from the pre-trained model. For a fair comparison, these baselines are trained on the union of source and target domain, as LogFormer utilize the knowledge of source domain (BGL) and target domain (HDFS/Thunderbird). For evaluation, we use Precision ($\frac{TP}{TP+FP}$), Recall ($\frac{TP}{TP+FN}$) and F_1 score ($\frac{2*Precision*Recall}{Precision+Recall}$).

Main Results In Table 2, baselines are trained on the union of source and target domain data for a fair comparison. In our setting, BGL dataset is chosen as the source domain for the pre-training, HDFS and Thunderbird are chosen as the target domain. LogFormer achieves the highest F_1 score on all three settings. Specifically, results show that most baseline methods perform badly when BGL data is used for training. It is reasonable for the diverse types of error and complex structure of logs in BGL. This also confirms that these baselines have poor generalizability and cannot handle multi-source logs together. When only BGL data is used for training, LogRobust and PLElog achieve a comparable F_1 score with LogFormer_S, this means our backbone model with Log-Attention module is strong enough without pre-training

³<https://github.com/HC-Guo/LogFormer>.

Method	HDFS		BGL		Thunderbird	
	Train	Test	Train	Test	Train	Test
DeepLog	50m	10m	23m	6m	59m	12m
LogAnomaly	1h 48m	22m	1h 10m	20m	1h 43m	30m
PLElog	42m	35s	20m	14s	36m	31s
LogRobust	1h 01m	17m	40m	4m	58m	12m
LogFormer	29m	20s	17m	11s	31m	16s

Table 3: Time consumption of different approaches. The lowest results are highlighted.

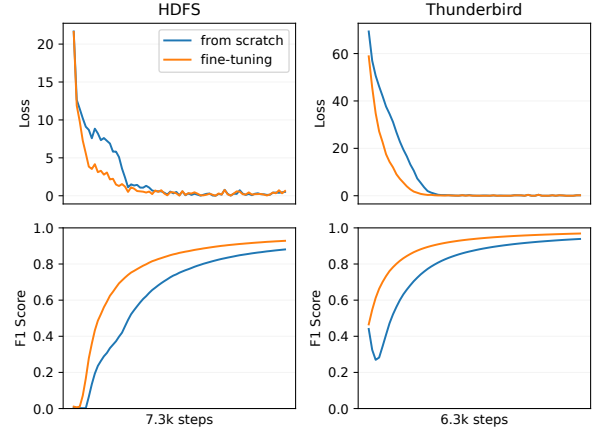


Figure 6: Loss and F_1 score on the test set. We compare two ways of training including training from scratch and fine-tuning from the pre-trained model on BGL.

and adapter-tuning on the single source.

Time Consumption Table 3 shows the training and testing time of LogFormer on HDFS, BGL, and Thunderbird, respectively. LogFormer gains the lowest training and testing time consumption compared with these state-of-the-art methods.

Ablation Study

Effect of pre-training To demonstrate the effectiveness of pre-training, we compare the performance of LogFormer_S and LogFormer_P. We choose BGL as the source domain as its variety in log templates. We compare two strategies in terms of loss and F_1 score. Fig 6 shows the loss and F_1 score curves in the training process (i.e., training steps). Results show that fine-tuning converges faster than training from scratch, which shows the learned knowledge from the source domain is valuable. Besides, the method with fine-tuning achieves higher performance in the initial stage and the loss curve is more stable, which also shows the power of pre-training. For the F_1 , we observe that fine-tuning requires fewer training steps to gain the best results, which is noteworthy for reducing costs in industrial scenes. To sum up, the pre-training stage is valuable and allows the model to converge quickly with better results.

Effect of Adapter-based Tuning Although we have shown that pre-training could accelerate convergence without de-

Method	#Layers	#Parameters	HDFS	Thunderbird
Tuning	1	7.2M	0.945	0.969
	2	14.3M	0.962	0.981
	4	28.5M	0.961	0.980
Adapter tuning	1	0.4M	0.957	0.972
	2	0.6M	0.974	0.987
	4	1M	0.981	0.998

Table 4: Results between fine-tuning and adapter-based tuning. #Layers is the number of encoder layers. #Parameters is the number of trainable parameters.

Attention	HDFS	BGL	Thunderbird
Self-Attention	0.911	0.943	0.935
Log-Attention	0.952	0.974	0.962

Table 5: F_1 scores between self-attention and Log-Attention. Experiments are based on 4 encoder layers.

creasing performance, fine-tuning is expensive and important. Thus, we adopt adapter-based tuning to acquire a compact model for log anomaly detection by adding a few additional trainable parameters. To show the effect of adapter-based tuning, we compare the performance of LogFormer_p and LogFormer on the HDFS and Thunderbird datasets as shown in Table 4. We have the following observations. First, LogFormer generates a little higher F_1 score (1% on average) than directly fine-tuning the pre-trained model on two datasets. Second, Adapter-based tuning adopts 3.5% – 5.5% of the trainable parameters compared to direct fine-tuning. Third, more encoder layers for fine-tuning do not generate better results. In contrast, adapter-based tuning performs more robustly with more encoder layers.

Effect of Log-Attention We compare the results of the original self-attention with our Log-Attention. To avoid the interference of other factors, we use LogFormer_S in Table 5. Results show that our Log-Attention achieves 3.6% higher points on average than the self-attention on three datasets, which shows the effect of Log-Attention module. Meanwhile, it shows that variables (removed by log parsing) also provide valuable information for the anomaly detection.

Effect of Variants of Adapters We compare the F_1 scores of the variants of adapters including LoRA (Hu et al. 2022) and Parallel-Adapter (Zhu et al. 2021) in Table 6. Results show that in our task, all three types of adapter gains great performance on three datasets, demonstrating the effectiveness of adapter-based tuning stage.

Effect on Low-resource Setting To verify the power of LogFormer under the low-resource setting, we consider the task with fewer than 20k training examples as the low-resource setting. The ablation study is conducted on the Thunderbird and models are sufficiently trained for 30 epochs. In Fig. 7, we compare the F_1 scores with different numbers of training samples ranging from 5k-20k. We find that 1) Adapter-based tuning consistently outperforms training and

Adapter	HDFS	BGL	Thunderbird
Serial Adapter	0.982	0.971	0.992
Parallel Adapter	0.980	0.969	0.988
LoRA	0.981	0.972	0.993

Table 6: F_1 scores between serial adapter and ours. Experiments are based on 4 layers of log-attention encoder.

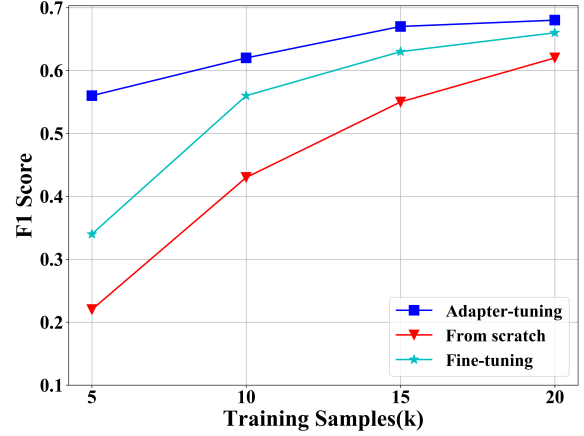


Figure 7: Test performance on Thunderbird w.r.t. the number of training examples. 5k, 10k, 15k, 20k corresponding to the first 1.25%, 2.5%, 3.75%, 5% training data respectively. We show F_1 scores for all methods.

fine-tuning, especially when the training size is small. For example, we gain 34% improvements compared with training from scratch with only 5k data. 2) With the number of training samples increasing, the gap between the F_1 scores of all methods will become smaller. 3) LogFormer is robust, with a similar standard deviation across different training sizes. To summarize, LogFormer provides acceptable results in the low-resource setting, which is highly parameter-efficient for log analysis.

Effect of Source Domain LogFormer aims at transferring knowledge between domains to help detect log anomalies. Thus it is vital to choose the correct source domain for the pre-training stage. A suitable domain needs to meet two conditions: 1) Variety in templates and types of error. 2) For different and similar domains, it has the great power to migrate semantic knowledge. HDFS has fewer templates and types of error compared with BGL and Thunderbird. Thus we do not utilize HDFS as the source domain. Specifically, we compare the results by choosing BGL and Thunderbird as the source domain respectively. In terms of the F_1 score, both of them gain high results on target domains. Thus we turn our attention to loss curves, Fig. 8 shows the loss curves on the target domains. Comparing two pre-trained models, on the HDFS dataset, the model pre-trained on BGL brings faster convergence. Besides, the model pre-trained on BGL brings faster convergence for Thunderbird than Thunderbird brings to BGL. Overall, BGL is the most suitable source domain for

Method	HDFS	BGL	Thunderbird
ChatGPT w/o LCoT	0.78	0.74	0.81
ChatGPT w/ LCoT	0.85	0.83	0.88
LogFormer	0.98	0.97	0.99

Table 7: F_1 between ChatGPT with LCoT and without LCoT.

transferring semantics across domains.

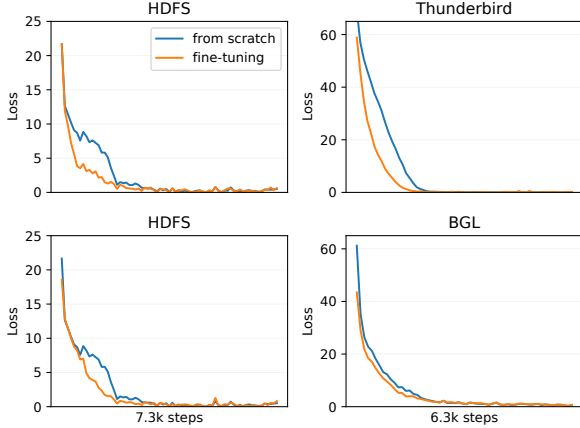


Figure 8: Loss on the test set w.r.t training steps. The upper/bottom results are based on models pre-trained on BGL/Thunderbird. All results are based on using one layer encoder.

Effect of Large Language Models To fully explore the ability of large language models to detect anomalous logs, we specially design a two-hop log Chain-of-Thought (Wei et al. 2022) (LCoT) approach for this task rather than directly generate the decision. Specifically, in Fig. 9, we first guide the model to generate the templates of the log sequences, then we concatenate the raw logs and templates as the prompts, and then let the model determine if there are any anomalies that arise. A comparison of the results using Chain-of-Thought and those generated directly is in Table. 7. We can see that large language models like ChatGPT can not perform well on log anomaly detection task. To explore why the large language model performs badly, we check the results and find that in most bad cases, the variables play an important role. Large language model is more concerned with sensitive words like "Error" and "failed". Such words contain obvious abnormalities, which can be recognized well. The findings further illustrate the importance of our log-attention, and in the future, we can inject external rules to make the large language models more sensitive to the variables.

Practical Evaluation

LogFormer has been successfully applied to a cloud service company. To test the generalization of LogFormer, we conduct experiments on a real-world distributed dataset called

Log Context	
<i>log sequences:</i>	2023-08-02 10:30:00 DEBUG: Checking server availability. 2023-08-02 10:30:15 ERROR: NetworkException - Unable to establish connection to server. 2023-08-02 10:31:00 INFO: Retrying connection to server.
Hop 1: Log Parsing	
<i>One-hop Prompt:</i>	Extract the templates of <i>log sequences</i> while replacing the <i>variables</i> with <code>< * ></code>
<i>Templates:</i>	1. <code>< * ></code> ERROR: NetworkException - <code>< * ></code> to establish connection to server. 2. <code>< * ></code> DEBUG: Checking server availability. 3. <code>< * ></code> INFO: Retrying connection to server.
Hop 2: Anomaly Detection	
<i>Two-hop Prompt:</i>	According to the <i>log sequences</i> , <i>Templates</i> ., the relationship between <i>Templates</i> : and <i>variables</i> , determine if there are any exceptions in templates and variables, and directly give the answer: Yes or No.
<i>Answer:</i>	Yes or No.

Figure 9: An example of two hop log Chain-of-Thought process. We first extract the templates of the logs, then we let the ChatGPT find the anomalies according to the logs and templates.

Dataset	Method	Precision	Recall	F_1 Score
GAIA	SVM	0.21	0.54	0.30
	DeepLog	0.18	0.82	0.31
	LogAnomaly	0.23	0.80	0.36
	PLELog	0.81	0.86	0.84
	LogRobust	0.83	0.94	0.88
	ChatGPT	0.68	0.75	0.71
	LogFormer	0.89	0.98	0.93

Table 8: Results of different methods on GAIA.

GAIA⁴. As the online system serves hundreds of corporations, the generated logs are complex. Thus, it is difficult to detect anomalies on such multi-domain and continuously evolved data. Here we take 8,200,000 log messages for the experiment (80% for training, 20% for testing), amounting to 31,279 anomalous messages. In Table 8, LogFormer still achieves the best performance among these baselines. Besides, LogFormer is stably running over 3000 hours on this system, which further demonstrates the stability of the model.

Conclusions

In this paper, we propose LogFormer, a pre-train and tuning pipeline for log anomaly detection, which contains the pre-training stage and the adapter-based tuning stage. Besides, the Log-Attention module is proposed to better encode the information of parameters. Extensive experiments show that our LogFormer, with fewer trainable parameters and lower training costs, outperforms all previous baselines.

⁴<https://github.com/CloudWise-OpenSource/GAIA-DataSet>

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 62276017, U1636211, 61672081), and the Fund of the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2021ZX-18).

References

- Bai, J.; Guo, H.; Liu, J.; Yang, J.; Liang, X.; Yan, Z.; and Li, Z. 2023. GripRank: Bridging the Gap between Retrieval and Generation via the Generative Knowledge Improved Passage Ranking. In *CIKM 2023*, 36–46. ACM.
- Breier, J.; and Branišová, J. 2015. Anomaly detection from log files using data mining techniques. In *Information Science and Applications*. Springer.
- Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL 2019*.
- Du, M.; and Li, F. 2016. Spell: Streaming parsing of system event logs. In *ICDM 2016*.
- Du, M.; Li, F.; Zheng, G.; and Srikumar, V. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS 2017*.
- Guo, H.; Guo, Y.; Yang, J.; Liu, J.; Li, Z.; Zheng, T.; Zheng, L.; Hou, W.; and Zhang, B. 2023a. LogLG: Weakly Supervised Log Anomaly Detection via Log-Event Graph Construction. In *DASFAA 2023*, volume 13946 of *Lecture Notes in Computer Science*, 490–501. Springer.
- Guo, H.; Liu, J.; Huang, H.; Yang, J.; Li, Z.; Zhang, D.; and Cui, Z. 2022. LVP-M3: Language-aware Visual Prompt for Multilingual Multimodal Machine Translation. In *EMNLP 2022*, 2862–2872. Association for Computational Linguistics.
- Guo, H.; Yang, J.; Liu, J.; Yang, L.; Chai, L.; Bai, J.; Peng, J.; Hu, X.; Chen, C.; Zhang, D.; Shi, X.; Zheng, T.; Zheng, L.; Zhang, B.; Xu, K.; and Li, Z. 2023b. OWL: A Large Language Model for IT Operations. *CoRR*, abs/2309.09298.
- He, P.; Zhu, J.; Zheng, Z.; and Lyu, M. R. 2017. Drain: An online log parsing approach with fixed depth tree. In *ICWS 2017*, 33–40.
- He, S.; Zhu, J.; He, P.; and Lyu, M. R. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *CoRR*, abs/2008.06448.
- Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 1735–1780.
- Houlsby, N.; Giurgiu, A.; Jastrzebski, S.; Morrone, B.; De Laroussilhe, Q.; Gesmundo, A.; Attariyan, M.; and Gelly, S. 2019. Parameter-efficient transfer learning for NLP. In *ICML 2019*.
- Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR 2022*.
- Huang, S.; Liu, Y.; Fung, C.; He, R.; Zhao, Y.; Yang, H.; and Luan, Z. 2020. HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log. *TNSM*, 17(4): 2064–2076.
- Jiang, Z. M.; Hassan, A. E.; Flora, P.; and Hamann, G. 2008. Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In *QSIC 2008*, 181–186.
- Lu, S.; Wei, X.; Li, Y.; and Wang, L. 2018. Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network. In *DASC 2018*, 151–158.
- Makanju, A.; Zincir-Heywood, A. N.; and Milios, E. E. 2009. Clustering event logs using iterative partitioning. In *KDD 2009*, 1255–1264.
- Meng, W.; Liu, Y.; Zhu, Y.; Zhang, S.; Pei, D.; Liu, Y.; Chen, Y.; Zhang, R.; Tao, S.; Sun, P.; et al. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *IJCAI 2019*.
- Oliner, A. J.; and Stearley, J. 2007. What Supercomputers Say: A Study of Five System Logs. In *DSN 2007*, 575–584.
- Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P. F.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. In *NeurIPS*.
- Reimers, N.; and Gurevych, I. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP 2019*, 3980–3990.
- Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Ichter, B.; Xia, F.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.
- Xu, W.; Huang, L.; Fox, A.; Patterson, D.; and Jordan, M. I. 2010. Detecting large-scale system problems by mining console logs. In *ICML 2010*.
- Yang, J.; Yin, Y.; Ma, S.; Zhang, D.; Wu, S.; Guo, H.; Li, Z.; and Wei, F. 2022. UM4: Unified Multilingual Multiple Teacher-Student Model for Zero-Resource Neural Machine Translation. In *IJCAI 2022*, 4454–4460. ijcai.org.
- Yang, L.; Chen, J.; Wang, Z.; Wang, W.; Jiang, J.; Dong, X.; and Zhang, W. 2021a. PLELog: Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation. In *ICSE 2021*, 230–231.
- Yang, L.; Chen, J.; Wang, Z.; Wang, W.; Jiang, J.; Dong, X.; and Zhang, W. 2021b. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *ICSE 2021*, 1448–1460.
- Yao, K.; Li, H.; Shang, W.; and Hassan, A. E. 2020. A study of the performance of general compressors on log files. *ESE*, 25(5): 3043–3085.
- Zhang, K.; Xu, J.; Min, M. R.; Jiang, G.; Pelechris, K.; and Zhang, H. 2016. Automated IT system failure prediction: A deep learning approach. In *BigData 2016*.
- Zhang, S.; Liu, Y.; Pei, D.; Chen, Y.; Qu, X.; Tao, S.; and Zang, Z. 2015. Rapid and robust impact assessment of software changes in large internet-based services. In *ENET 2015*.
- Zhang, X.; Xu, Y.; Lin, Q.; Qiao, B.; Zhang, H.; Dang, Y.; Xie, C.; Yang, X.; Cheng, Q.; Li, Z.; et al. 2019. Robust log-based anomaly detection on unstable log data. In *FSE 2019*.

Zhu, Y.; Feng, J.; Zhao, C.; Wang, M.; and Li, L. 2021. Serial or Parallel? Plug-able Adapter for multilingual machine translation. *CoRR*.