*Article*

# Leveraging Large Language Models and BERT for Log Parsing and Anomaly Detection

**Yihan Zhou [1], Yan Chen [2], Xuanming Rao [1], Yukang Zhou [3], Yuxin Li [4] and Chao Hu [5,*]**

1 School of Computer Science and Engineering, Central South University, Changsha 410083, China; 8209210529@csu.edu.cn (Y.Z.); 8212210903@csu.edu.cn (X.R.)
2 Logistics Department, Central South University, Changsha 410083, China; 37050006@csu.edu.cn
3 Department of Electrical and Information Engineering, Hong Kong Polytechnic University, Hong Kong, China; 23050384g@connect.polyu.hk
4 School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore; yuxin.li@ntu.edu.sg
5 School of Electronic Information, Central South University, Changsha 410083, China
* Correspondence: huchao@csu.edu.cn

**Abstract:** Computer systems and applications generate large amounts of logs to measure and record information, which is vital to protect the systems from malicious attacks and useful for repairing faults, especially with the rapid development of distributed computing. Among various logs, the anomaly log is beneficial for operations and maintenance (O&M) personnel to locate faults and improve efficiency. In this paper, we utilize a large language model, ChatGPT, for the log parser task. We choose the BERT model, a self-supervised framework for log anomaly detection. BERT, an embedded transformer encoder, with a self-attention mechanism can better handle context-dependent tasks such as anomaly log detection. Meanwhile, it is based on the masked language model task and next sentence prediction task in the pretraining period to capture the normal log sequence pattern. The experimental results on two log datasets show that the BERT model combined with an LLM performed better than other classical models such as Deelog and Loganomaly.

**Keywords:** BERT; anomaly log detection; transformer; self-attention; large language models

**MSC:** 68T50

**Citation:** Zhou, Y.; Chen, Y.; Rao, X.; Zhou, Y.; Li, Y.; Hu, C. Leveraging Large Language Models and BERT for Log Parsing and Anomaly Detection. *Mathematics* **2024**, *12*, 2758. https://doi.org/10.3390/math12172758

Academic Editor: Cheng-Chi Lee

Received: 19 July 2024
Revised: 30 August 2024
Accepted: 3 September 2024
Published: 5 September 2024

## 1. Introduction

Computer systems and applications generate a large amount of logs to measure and record information during monitoring. System log data is one of the most valuable types of data. When security vulnerabilities occur, logs may be the best line of defense [1]. As systems' complexity and scale continue to increase, anomalies have become inevitable. Log anomaly detection (LAD) monitors logs in real time to identify and recommend root causes, helping operations and maintenance personnel pinpoint faults and improve operational efficiency. Therefore, anomaly detection is essential for quality assurance in complex, software-intensive systems.

Traditional log anomaly detection methods rely on manually written regular expressions or template patterns to parse logs. While these methods are simple, manual log analysis for anomaly detection becomes extremely time-consuming and error-prone when dealing with large-scale distributed systems and frequent update requirements. Therefore, automatic log parsing has emerged as a solution. To achieve the goal of automatic log parsing, numerous methods have been proposed in academia and industry, including frequent pattern mining (SLCT [2] and its extension LogCluster [2]), iterative partitioning (IPLoM [2]), hierarchical clustering (LKE [2]), longest common subsequence calculation (Spell [3]), and parse trees (Drain [4]). Subsequently, log event features are extracted using identifier-based or timestamp-based methods, and machine learning techniques, either supervised or unsupervised, are employed to detect system anomalies.

In recent years, the application of deep learning in the field of anomaly detection has gradually attracted attention, among which the DeepLog model, LogAnomaly model, and LogRobust model based on LSTM has achieved remarkable results in log anomaly detection. LSTM [5] is a special type of recurrent neural network (RNN) that can process sequence data efficiently. The DeepLog [6] model uses LSTM to encode system logs and capture timing information and semantic relationships in logs. By training normal log data, the DeepLog model can learn patterns of normal behavior and compare abnormal behavior with normal behavior to detect abnormal events. To solve the problem that Deeplog only uses the log template index, which may lose valuable semantic information, LogAnomaly is proposed. LogAnomaly [7] is a framework that automatically detects both sequential and quantitative anomalies end-to-end using LSTM networks. Inspired by word embedding, template2vec is proposed in LogAnomaly to capture semantic information including synonyms and antonyms. log robust group templates with the same features, in HDFS data, that is, templates with the same blkId, together for the model to learn its features. Log Robust [8] Extracts semantic information of log events and represents it as a vector. Anomalies are detected using an attention-based Bi-LSTM model that captures contextual information in log sequences and automatically understands the significance of different log events.

However, all of these methods rely on log parsing to preprocess semi-structured log data. Through empirical studies on real log data, we found that traditional log parsing methods have significant shortcomings when processing the large volumes of logs generated by distributed systems. By analyzing the incorrect parsing results of Drain and Spell, we discovered that: (1) Drain relies on fixed templates for log parsing, which may fail to accurately parse unknown or new log patterns, resulting in template matching failures or mismatches. Drain also ignores the contextual relationships between logs and cannot capture more complex log dependencies. (2) Similarly, Spell, which is based on word frequency statistics for log parsing, performs poorly with low-frequency or sudden anomalous logs and is less effective in parsing complex or diverse log patterns.

In order to address the limitations mentioned earlier in existing methods, this study performs log anomaly detection tasks based on structured logs parsed by large language models. This approach benefits from the rich features in the parsed log data and retains the contextual information of the logs. Large language models can effectively filter out noise in the logs, extract key features, and reduce false positives and false negative rates. We use both zero-shot learning and few-shot learning methods for training. Zero-shot learning enables ChatGPT to abstract variables from the provided log messages and generate the corresponding templates. Enhanced zero-shot learning allows ChatGPT to identify and abstract all dynamic variables in the logs and output a static log template. Few-shot learning provides some examples to help ChatGPT learn how to abstract variables from log messages and generate the corresponding templates. We group the raw logs and perform multi-threaded template extraction on the groups. Finally, we post-process the log templates parsed by the large language model, replacing the variable names identified in the templates with placeholders. We use TF-IDF for feature extraction by calculating the term frequency and inverse document frequency to identify distinctive feature words in the logs. This helps in identifying anomalous log entries, thereby assisting in anomaly detection and troubleshooting. We use the BERT model for anomaly detection tasks, leveraging the pre-training capabilities of large language models to provide better initial feature representations for BERT, thereby improving the model's learning efficiency and effectiveness.

We have evaluated the proposed method using two public datasets. The experimental results show that structured logs parsed based on large language models can understand the semantic meaning of log data and effectively handle contextual information, providing low-noise and accurate input for downstream log anomaly detection tasks. It achieved high F1 scores, precision, and recall (all greater than 0.90) in anomaly detection on the BGL dataset, outperforming existing log-based anomaly detection methods.

Our main contributions can be summarized as follows.

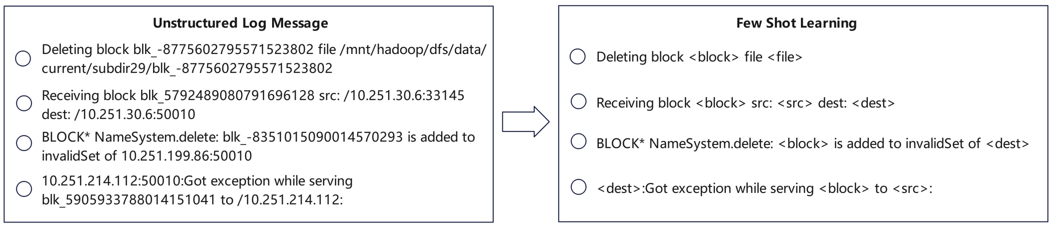
- We conducted an empirical study on the accuracy of log parsing. We found that existing parsers like Drain, Spell, and AEL struggle to capture the context between logs and tend to have lower message-level parsing accuracy when dealing with complex logs, leading to adverse effects such as parsing errors and semantic misunderstandings.
- We use sliding window event counts combined with TF-IDF weights to learn the representation vectors of log templates generated by the large language model.
- We use large language models combined with few-shot learning to capture log semantic information for log parsing. We employ LogBERT, a deep learning-based Transformer network, to encode the semantics of log messages and identify normal log sequences.
- We evaluated the log parsing capability of large language models using public datasets. The results confirmed the effectiveness of LLMs in log parsing, particularly in improving log message accuracy and enhancing the precision of downstream log anomaly detection.

## 2. Related Work

### *2.1. Log Parsing*

The log parsing module, the first research point of this paper, is an upstream task of the overall process of ensuring system reliability. It is mainly intended to extract effective information from the original text structure log and parse it into a structured log which can be recognized by the downstream task (exception detection module). The traditional log parsing method relies too much on the domain knowledge of the system's operation and maintenance personnel and is not suitable for the fast iteration of modern server application scenarios. Traditional log parsers lack attention to semantic information. SemParser [9], studied by Yintong Huo et al., is a semantic parser for log analysis. It can focus on the semantic association between two information templates. The template semantics of logs can be obtained independently through the semantic miner, with implicit semantic reasoning conducted using the joint parser. Finally, LSTM and softmax functions are constructed in the semantic miner to mine the semantics of word vectors. Brain [10], studied by Siyu Yu, Pinjia He et al., is a log parser based on bidirectional parallel trees. Inspired by the longest common pattern between logs possibly being a log template, he created initial groups based on the longest common pattern in the birthday logs. The bidirectional tree is used to supplement the constant words to the longest common pattern, thus forming a complete log template effectively. In our experiments, we use AEL, Drain [4], and ChatGPT to parse the raw log.

In our log parser section, we not only use traditional log parsers like Drain, AEL, and Spell [3] for structuring raw logs, but we also introduce OpenAI's large language models [11] to identify log templates by learning the patterns and structures of log data. Large language models, such as GPT-3 or GPT-4, possess powerful contextual understanding abilities which enable them to recognize and parse complex log entries. This capability is particularly beneficial when dealing with logs which have multi-level nested structures and multi-line entries. Furthermore, large language models can handle various log formats and structures without the need for predefined, strict templates, offering greater flexibility and adaptability when faced with different types of log files. Lastly, these models can learn new log patterns with a small number of samples, allowing them to quickly adapt to changing log formats and emerging log types. Few-shot learning [12] is a machine learning approach aimed at building accurate models from very few training data. It can be divided into zero-shot learning and one- or few-shot learning, which aim to recognize unseen categories and categories with only one or a few samples, respectively. In contrast, traditional parsers often require extensive manual annotation and template definition. In our experiments, we set up four control groups: zero-shot learning, one-shot learning, two-shot learning, and four-shot learning. A four-shot example is shown in Figure 1. The experiments demonstrated that the more samples provided, the higher the accuracy in recognizing log templates.



**Unstructured Log Message**

```
○ Deleting block blk_-8775602795571523802 file /mnt/hadoop/dfs/data/
  current/subdir29/blk_-8775602795571523802
○ Receiving block blk_5792489080791696128 src: /10.251.30.6:33145
  dest: /10.251.30.6:50010
○ BLOCK* NameSystem.delete: blk_-8351015090014570293 is added to
  invalidSet of 10.251.199.86:50010
○ 10.251.214.112:50010:Got exception while serving
  blk_5905933788014151041 to /10.251.214.112:
```

**Few Shot Learning**

```
○ Deleting block <block> file <file>
○ Receiving block <block> src: <src> dest: <dest>
○ BLOCK* NameSystem.delete: <block> is added to invalidSet of <dest>
○ <dest>:Got exception while serving <block> to <src>:
```

**Figure 1.** Four-shot learning of ChatGPT on HDFS dataset.

## 2.2. *Log Partitioning and Feature Extraction*

Logs are textual messages which need to be converted into numerical features for machine learning algorithms to understand them. First, a log parser is used to identify log templates to represent each log message. Then, log timestamps and log identifiers (such as task, job, or session IDs) are typically used to partition logs into different groups, with each group representing a log sequence. Log partitioning based on timestamps generally includes two strategies: fixed partitioning and sliding partitioning.

Fixed partitioning has a predefined partition size, representing the time interval used to split the chronologically ordered logs. In this case, there is no overlap between two consecutive fixed partitions. For example, if the partition size $\triangle t$ is one hour or even one day, the logs will be divided into multiple non-overlapping parts based on this time interval.

Sliding partitioning involves two parameters: the partition size and stride. The stride represents the distance the time window moves along the time axis to generate log partitions. Typically, the stride is smaller than the partition size, resulting in overlaps between different sliding partitions. For instance, if the partition size is $\triangle t$, and the stride is $\frac{\triangle t}{3}$, then more log sequences will be generated, with some logs appearing in multiple partitions.

Identifier-based partitioning sorts logs chronologically and divides them into different sequences based on a specific identifier (such as a task ID or session ID). In each sequence, all logs share a unique identifier, indicating that they originate from the same task execution. For example, HDFS logs use block identifiers to record operations related to a specific block, such as allocation, replication, and deletion. Log sequences generated this way usually vary in length, with shorter sequences possibly due to premature termination caused by anomalies.

After log partitioning, many traditional machine learning-based methods generate input features by creating a log event count vector, where each dimension represents a log event and its value indicates the frequency of that event in the log sequence. In contrast, deep learning-based methods often use log event sequences directly. Each element in the sequence can simply be the index of a log event or more complex features such as log embedding vectors. To learn the semantics of logs, words in log events are first represented by word embeddings learned using algorithms like word2vec, FastText, and GloVe. These word embeddings are then aggregated to form the semantic vector of the log event. These vectors aim to capture the semantics of the logs for making more intelligent decisions.

## 2.3. *Anomaly Detection*

After extracting log features, one can perform exception detection (i.e., identify abnormal instances in logs). Based on the focus of data parsing, anomaly detection is divided into two categories: log anomaly detection and time series anomaly detection. Log anomaly detection typically deals with unstructured or semi-structured text data, such as system events and operation records. Log data usually consists of fields like timestamps, event types, and message contents. Logs record discrete, often non-periodic events, which may not have clear time series characteristics. In contrast, time series anomaly detection aims to identify anomalous points, intervals, or patterns within sequential data, which may indicate issues with data collection, system failures, or abnormal behavior.

#### 2.3.1. Time Series Anomaly Detection

The aim of time series anomaly detection is to spot data points which significantly differ from the majority within a time series. Anomalies are often rare and mixed with a large number of normal points, which makes labeling the data a challenging task. Therefore, most research focuses on using unsupervised methods to identify anomalies. In recent years, deep learning techniques have shown great promise in anomaly detection due to the ability of neural networks to learn representations. However, when anomalies are not evenly spread across the entire time series but are concentrated in specific regions—a phenomenon known as anomaly concentration—neural network-based methods may struggle to accurately identify these anomalies. This is because their models are usually trained on regions dominated by normal data.

Time series anomaly detection can be divided into two methods. One is unsupervised time series anomaly detection, which learns information representations from complex temporal dynamics through unsupervised tasks. The other is based on explicit association modeling to detect anomalies. To deal with the anomaly concentration, Xiao et al. [13], among others, recently proposed a diffusion-based time series anomaly detection framework to mitigate performance degradation under an anomaly concentration. By selecting a small set of discrete points to estimate errors, the influence of dense anomaly points is reduced. Xu et al. [14], among others, applied transformers to unsupervised time series anomaly detection, enhancing the distinction between normal and abnormal associations.

#### 2.3.2. Log-Based Anomaly Detection

Over the years, many log-based anomaly detection approaches have been proposed. Some of them are based on unsupervised learning methods, which require only unlabeled data to train. For example, Xu et al. [15] employed principal component analysis (PCA) to generate two subspaces (normal space and anomaly space) of log count vectors. If a log sequence has its log count vector far from the normal space, then it is considered an anomaly. There are also many supervised anomaly detection approaches. For example, one can represent log sequences as log count vectors and then apply a support vector machine (SVM), logistic regression (LR), or decision tree algorithm to detect anomalies. These approaches have many common characteristics. They all require a log parser to preprocess and extract log templates from log messages. Then, the occurrences of log templates are counted, resulting in log count vectors. Finally, a machine learning model is constructed to detect anomalies.

In recent years, many deep learning-based models have been proposed to analyze log data and detect anomalies. Mainstream deep learning models encompass LSTM-based algorithms such as DeepLog, LogAnomaly, and Autoencoder, which fall under the category of unsupervised learning methods. Additionally, there are supervised learning-based models like LogRobust and the CNN. DeepLog [6] first uses the Spell parser to extract log templates. It then utilizes the indexes of these log templates and inputs them into an LSTM model to predict the next log templates. Finally, DeepLog detects anomalies by determining whether the incoming log templates are unexpected. LogAnomaly [7] uses log count vectors to detect anomalies reflected by anomalous log event numbers. It proposes a method based on synonyms and antonyms to represent the words in log templates. LogRobust [8] incorporates a pretrained Word2vec model, namely FastText, and combines it with TF-IDF weights to learn the representation vectors of log templates generated by Drain. These vectors are then input into an attention-based Bi-LSTM model to detect anomalies. Due to the imperfections in log parsing, these methods often lose the semantic meaning of log messages, leading to inaccurate detection results. Inspired by DeepLog and LogAnomaly, we segment logs to form log sequences and use log event vectors as inputs to a deep neural network. Following the Autoencoder approach, we extract features from log events through an encoder. The introduction of LogRobust made us aware of the importance of the attention mechanism, which can capture the context of each log entry. Therefore, the anomaly detection model proposed in this paper is based on the BERT model. BERT uses a Transformer encoder to learn the contextual relationships of log templates in sequences. By leveraging two self-supervised training tasks, BERT learns the patterns of normal log sequences and can detect anomalies where underlying patterns deviate from the normal log sequences.

## 3. Proposed Framework

### *3.1. Large Language Models for Log Parser*

In this section, we introduce how to use large language models for log parsing. We chose the ChatGPT-3.5 model from OpenAI. Compared with other large language models, ChatGPT adopts the Transformer architecture, capable of capturing long-range contextual relationships. This enables it to better understand the correlations between various parts of the log and parse log information more accurately. Additionally, ChatGPT is a versatile conversational model with strong flexibility and adaptability, allowing it to handle different types and formats of log data. We divide the explanation into three sections: parameter file configuration, log template parsing, and template post-processing.

#### 3.1.1. Parameter File Configuration

The parameter file defines various configurations needed for interacting with the GPT model and is closely related to the parsing results. We divided the parameter file into three modules: basic configuration, zero-shot learning parameters, and few-shot learning parameters.

Table 1 shows the key information in the basic configuration. Among them, an Open AI key was used to access the API. model, which was used to specify a GPT model like GPT-turbo-1106. The temperature, frequency penalty, and presence penalty were used to adjust the model's output style and quality during text generation, ensuring that the generated text better met specific requirements and expectations.

**Table 1.** Basic configuration.

| **Parameter Name** | **Description** |
|---|---|
| OPEN_AI_KEY | Key used to access the OpenAI API |
| MODEL | Specifies the GPT model to use, which is GPT-3.5-turbo-1106 here |
| TEMPERATURE | Controls the randomness of generated text, with higher values (up to 1.0) resulting in more randomness, while 0.0 indicates complete determinism |
| FREQUENCY_PENALTY | Controls the penalty for repeated word usage, with higher values making the model more inclined to generate a more diverse set of words |
| PRESENCE_PENALTY | Controls the inclination to generate new words, with higher values making the model more inclined to generate new words rather than repeated ones |

Table 2 shows the key information in the zero-shot learning configuration. This prompt template was used for zero-shot baseline testing. In this case, ChatGPT needed to abstract variables from the provided log message and generate the corresponding template. The template represents variables with *placeholders*.

**Table 2.** Zero-shot learning configuration.

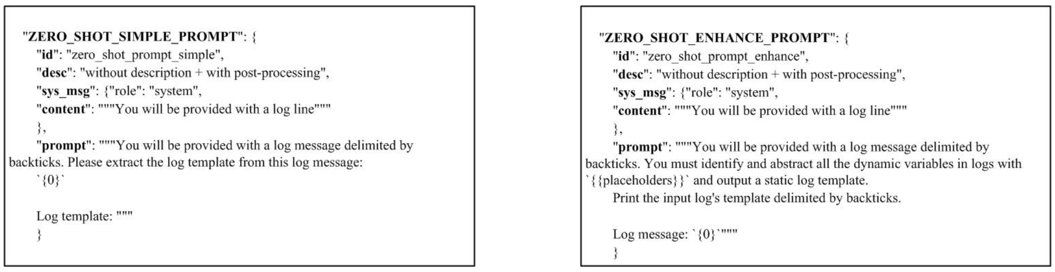
| Parameter Name | Description |
|---|---|
| Sys_Msg | Sets the initial state of the model, specifying that the user will provide a log message |
| Prompt | Template message used to guide the model in generating a log template, requiring the model to abstract variables in the log message as placeholders |

Table 3 shows the key information in the few-shot learning configuration. This prompt template was used for few-shot baseline testing. It provided some examples to help ChatGPT learn how to abstract variables from log messages and generate the corresponding template. The template represents variables with *placeholders*.

**Table 3.** Few-shot learning configuration.

| Parameter Name | Description |
|---|---|
| demo_format | Format when providing examples, which here is the template of 0is1 |
| demo_instruct | Instruction provided before examples |
| Prompt | Template message asking the model to generate a log template |

Additionally, there are simplified template messages, requests for the model to directly extract templates from log messages, and enhanced template messages, requiring the model to recognize and abstract all dynamic variables in the logs and generate static log templates. The simplified prompt template and enhanced prompt template are shown in the following Figure 2:



```
"ZERO_SHOT_SIMPLE_PROMPT": {
    "id": "zero_shot_prompt_simple",
    "desc": "without description + with post-processing",
    "sys_msg": {"role": "system",
    "content": """You will be provided with a log line"""
    },
    "prompt": """You will be provided with a log message delimited by
backticks. Please extract the log template from this log message:
    `{0}`

    Log template: """
    }
```

```
"ZERO_SHOT_ENHANCE_PROMPT": {
    "id": "zero_shot_prompt_enhance",
    "desc": "without description + with post-processing",
    "sys_msg": {"role": "system",
    "content": """You will be provided with a log line"""
    },
    "prompt": """You will be provided with a log message delimited by
backticks. You must identify and abstract all the dynamic variables in logs with
`{{placeholders}}` and output a static log template.
    Print the input log's template delimited by backticks.

    Log message: `{0}`"""
    }
```

(**a**) (**b**)

**Figure 2.** Two types of zero-shot prompt template. (**a**) Zero-shot simple prompt template. (**b**) Zero-shot enhance prompt template.

### 3.1.2. Log Template Parsing

Log template parsing includes two types of benchmarks: zero-shot baseline testing and few-shot baseline testing. However, the basic operations are the same. First, the ChatGPT object is initialized, and the test log messages are load. Then, the log messages are partitioned into chunks, processing *MSG_LEN* messages at a time. By using multi-threading to parallel process each log chunk, one can obtain the log templates. Finally, the generated templates are saved to a log file, and the occurrences of each template are tallied before saving the results to a CSV file. The core step is the processing of each log chunk, which involves calling the @retry decorator to set up a retry mechanism. If the request fails, then it will be retried up to five times within a certain time frame, allowing the OpenAI API call to retrieve a response. Subsequently, regular expressions are used to batch extract templates from the response. If a template is found, then the last template and the original response are returned. If no template is found, then a warning log is recorded, and the original response is returned. The process is shown in Figure 3.

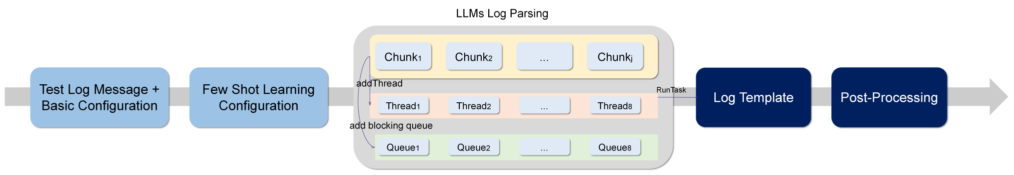


**Figure 3.** The process of log parser by an LLM.

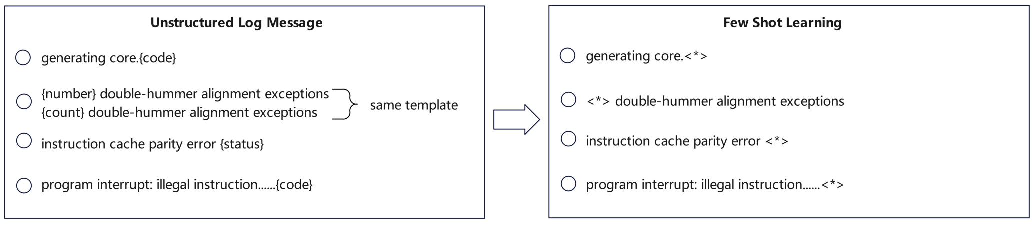
3.1.3. Template Post-Processing

The main purpose of post-processing is to read the log file and standardize its templates by applying a series of replacement rules which replace the variable parts with a common placeholder <*>. The final output is a processed log template file and a template statistics file for subsequent analysis and use. This process is divided into three steps.

Define the Template Processing Function: The function applies a series of rules to the input log template, replacing specific patterns with the common placeholder <*>. It handles eight specific patterns: double space (DS), Boolean (BL), user string (US), digit (DG), path-like string (PS), word concatenated with variable (WV), dot-separated variables (DVs), and consecutive variables (CVs).

Initialize Rules: This step defines some default Boolean values, default strings, and path separators.

Apply Rules to Process the Templates. (1) Remove Extra Spaces: The leading and trailing spaces are trimmed from the template, and any extra spaces are replaced with a single space. (2) Handle Path-Like Strings: The template is split by path separators, and matched path-like strings are replaced with <*>. (3) Handle Remaining Rules: The template is split by all separators, and then Boolean values, default strings, digits, and concatenated variable rules are applied to replace them accordingly. (4) Handle Consecutive Variables: Dot-separated and non-separated consecutive variables are replaced with <*>, and special cases like <*> and <*>:<*> are addressed. The results of post-processing are shown in Figure 4.



```
generating core.{code}
{number} double-hummer alignment exceptions
{count} double-hummer alignment exceptions
instruction cache parity error {status}
program interrupt: illegal instruction......{code}
```

```
generating core.<*>
<*> double-hummer alignment exceptions
instruction cache parity error <*>
program interrupt: illegal instruction......<*>
```

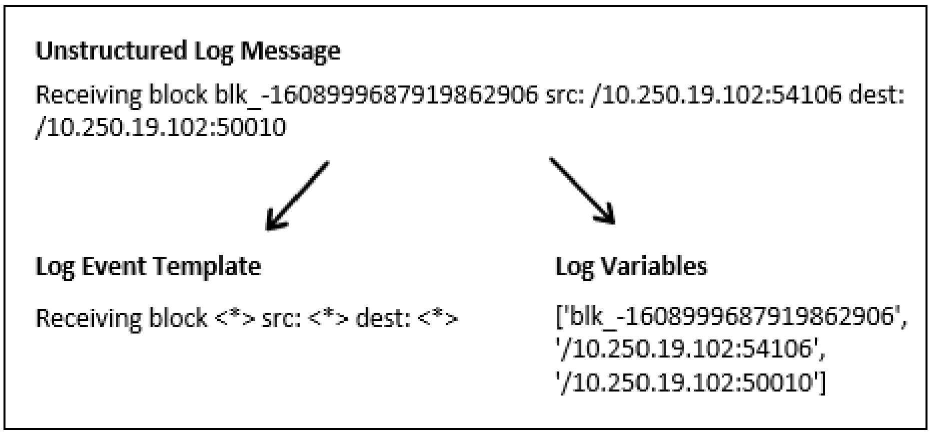
**Figure 4.** The results of post-processing.

*3.2. Feature Extraction*

The main purpose of log event feature extraction is to construct the feature data which can be processed by the machine learning model to learn the normal or abnormal patterns of the log. The quality of the extracted features determines the accuracy of the subsequent model detection effect. In order to automatically analyze the logs through a deep model, the log text in the log partition needs to be converted into a format suitable for the machine learning algorithm. Two types of log-based features are generally used, namely digital features and graphical features. There are four types of digital features. (1) The log event sequence is a sequence of log events which records system activities. (2) The log event count vector is the feature vector which records the occurrence of log events in the log partition. Each feature represents a log event type. The value counts the occurrence times. (3) The parameter value vector records the parameter values which appear in the log. (4) The ad hoc features are a set of relevant and representative features extracted from the log which are defined using domain knowledge of the object software system and the problem context. Graphical features typically produce a directed graph model which characterizes system behavior, discovering hierarchical and sequential relationships between the system components and the events and logs. The following describes how to extract the log event features of HDFS log datasets.

### 3.2.1. Obtain Log Structure

Suppose we have the original log text information in Figure 5. Semi-structured log information contains variables such as the timestamp, event level, block number, IP address, and constants. The log parsing algorithm needs to extract semi-structured log information to form log events; that is, it needs to parse the preceding examples into an event template ($Receiving\ Block < * > src : / < * > dest : / < * >$), event ID ($E$13), and content, among other elements. Among these, we only used the content, event ID, and variables. Note that the $Block < * >$ parameter $blk\_1608999687919862906$ identifies an anomaly label. We used regular expression to extract the block parameter in the content and build key and value pairs according to the event ID of the log to generate structured array objects. The log parser process is shown in Figure 5.



```
Receiving block blk_-1608999687919862906 src: /10.250.19.102:54106 dest:
/10.250.19.102:50010
```

```
Receiving block <*> src: <*> dest: <*>
```

```
['blk_-1608999687919862906',
'/10.250.19.102:54106',
'/10.250.19.102:50010']
```

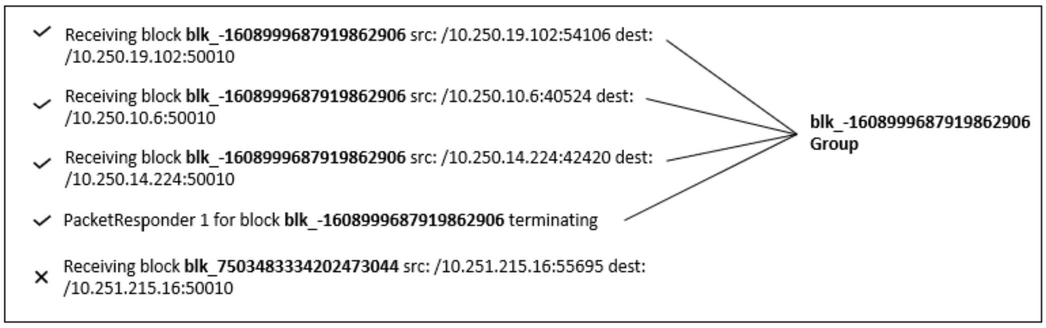
**Figure 5.** Parse the unstructured log message into event template and log variables.

### 3.2.2. Log Key Encoding and Log Parameter Encoding

The log key indicates the type of log, which is directly encoded by a sequential number. For the HDFS log dataset in this experiment, 40 different log templates could be sequentially encoded in the log template file parsed by the log parser. Unlike the log types, the parameter values are not generated by templates but dynamically generated according to actual situations during system operation, and thus they are often highly uncertain.

### 3.2.3. Log Partition

Modern software systems often adopt a microservice architecture and contain a large number of modules which run in a multi-threaded environment. Different microservices or modules often aggregate their execution logs into a single log file, which hinders automated log mining. To solve this problem, interlaced logs should be divided into different groups, each representing the execution of a single system task. The partitioned log group is the basic unit for extracting features before building the log mining model. It usually uses the timestamp and log identifier as the log divider. Since the HDFS log dataset has an abnormal-label.csv file, it divides normal or abnormal labels based on the value of the block ID. It can be seen that the parameter value of Block <*> has a direct relationship with log anomaly detection. Therefore, we divided logs with the same Block <*> parameter value into a log group and counted the log template type which each log group contained (i.e., Event ID). The model was trained by generating a feature matrix based on the log group The log partition process is shown in Figure 6.



```
✓ Receiving block blk_-1608999687919862906 src: /10.250.19.102:54106 dest:
  /10.250.19.102:50010
✓ Receiving block blk_-1608999687919862906 src: /10.250.10.6:40524 dest:
  /10.250.10.6:50010
✓ Receiving block blk_-1608999687919862906 src: /10.250.14.224:42420 dest:
  /10.250.14.224:50010
✓ PacketResponder 1 for block blk_-1608999687919862906 terminating
✕ Receiving block blk_7503483334202473044 src: /10.251.215.16:55695 dest:
  /10.251.215.16:50010
```

**Figure 6.** Log partition according to Block <*> parameter.

### 3.2.4. Event Count and TF-IDF Value Vectors

The event count indicates the count of the occurrence times of each log template in the log groups, divided according to the Block <*> parameter. Thus, the count vectors are constructed in the order of the log template. Then, we focused on the TF-IDF [16] model. Term frequency-inverse document frequency (TF-IDF) is a commonly used weighting technique for information retrieval and data mining. The main idea of TF-IDF is that if a word or phrase has a high TF in one article and rarely appears in other articles, then it is considered that this word or phrase has good classification ability and is suitable for classification. TF is the term frequency, and IDF is the inverse document frequency. We define the word frequency $tf_{t,d}$ as the frequency with which a keyword $t$ appears in a document $d$. In general, the higher the word frequency of a keyword in a document, the higher the relevance of the document to the keyword. We can measure this positive correlation using a logarithmic function:

$$w_{t,d} = \begin{cases} 1 + \log tf_{t,d} \ , if \ tf_{t,d} > 0 \\ 0 \qquad\qquad\quad , otherwise \end{cases} \tag{1}$$

where $tf_{t,d}$ denotes the log frequency weighting. In addition to the word frequency, the scarcity of keywords will also affect the matching degree of the search. Therefore, we hoped to give a greater weight to the rarer keywords and a lower weight to the more common keywords when searching. We defined $df_t$ as the number of documents in the dataset which contain the keyword $t$. Here, $df_t$ is inversely correlated with the information content of keyword $t$, and thus we used its inverse (inversed document frequency (idf)) to measure the importance of keywords:

$$idf_t = \log_{10} \frac{N}{df_t} \tag{2}$$

By combining the concepts of the TF and IDF, we obtained the TF-IDF model:

$$w_{t,d} = (1 + \log tf_{t,d}) \times \log_{10} \frac{N}{df_t} \tag{3}$$

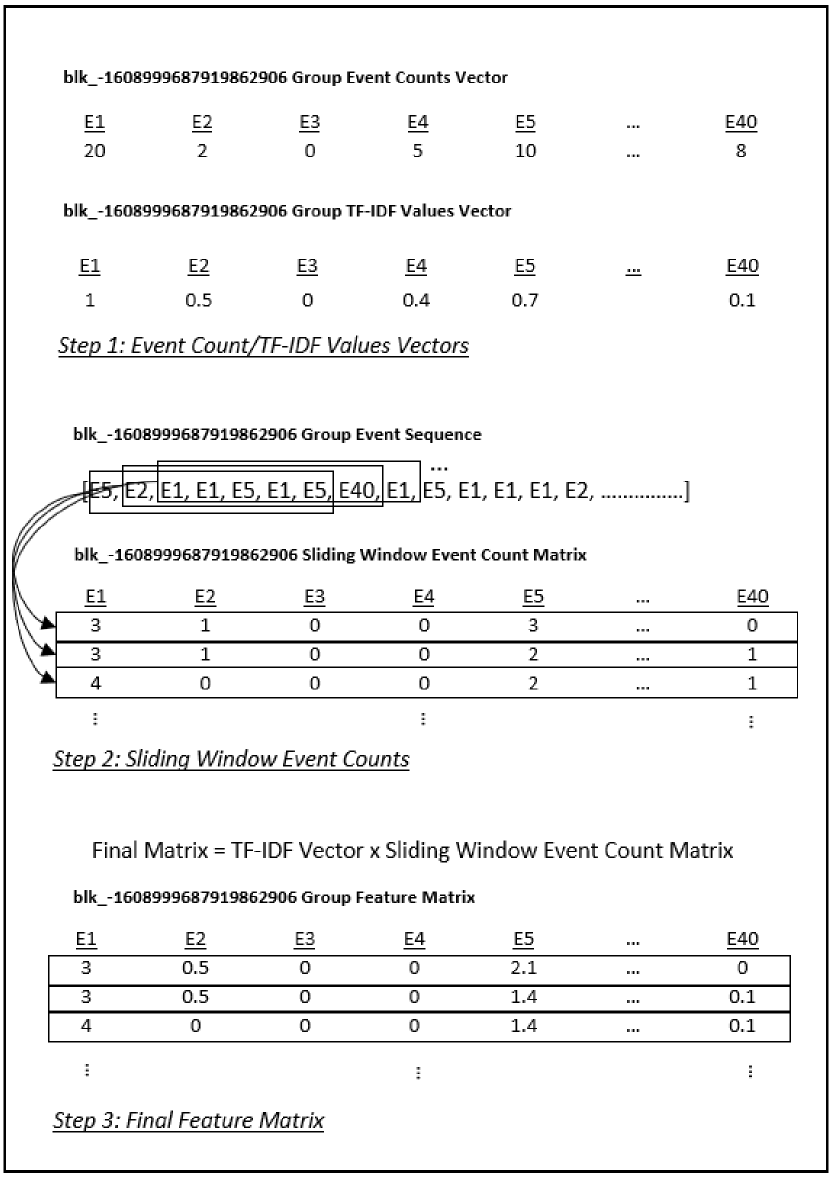
In simple terms, TF-IDF is the product of two scores, tf and idf, taking into account the word frequency and word scarcity.

### 3.2.5. Sliding Window Event Count and Final Feature Matrix

The sliding window event count [17] is a set window size and moves in a sequence according to a fixed stride. Each move extracts a row of data to form a row of the matrix and finally generate a sliding window event count matrix. The final feature matrix considers the TF-IDF value vector and event count vector together. We formulate it as follows:

$$Final\ Matrix = TF - IDF\ Vector \times Sliding\ Window\ Event\ Count\ Matrix$$

The calculation process of the final matrix is shown in Figure 7.



**blk_-1608999687919862906 Group Event Counts Vector**

| E1 | E2 | E3 | E4 | E5 | ... | E40 |
|---|---|---|---|---|---|---|
| 20 | 2 | 0 | 5 | 10 | ... | 8 |

**blk_-1608999687919862906 Group TF-IDF Values Vector**

| E1 | E2 | E3 | E4 | E5 | ... | E40 |
|---|---|---|---|---|---|---|
| 1 | 0.5 | 0 | 0.4 | 0.7 | | 0.1 |

*Step 1: Event Count/TF-IDF Values Vectors*

**blk_-1608999687919862906 Group Event Sequence**

...

[E5, E2, E1, E1, E5, E1, E5, E40, E1, E5, E1, E1, E1, E2, ...............]

**blk_-1608999687919862906 Sliding Window Event Count Matrix**

| E1 | E2 | E3 | E4 | E5 | ... | E40 |
|---|---|---|---|---|---|---|
| 3 | 1 | 0 | 0 | 3 | ... | 0 |
| 3 | 1 | 0 | 0 | 2 | ... | 1 |
| 4 | 0 | 0 | 0 | 2 | ... | 1 |
| ⋮ | | | ⋮ | | | ⋮ |

*Step 2: Sliding Window Event Counts*

Final Matrix = TF-IDF Vector x Sliding Window Event Count Matrix

**blk_-1608999687919862906 Group Feature Matrix**

| E1 | E2 | E3 | E4 | E5 | ... | E40 |
|---|---|---|---|---|---|---|
| 3 | 0.5 | 0 | 0 | 2.1 | ... | 0 |
| 3 | 0.5 | 0 | 0 | 1.4 | ... | 0.1 |
| 4 | 0 | 0 | 0 | 1.4 | ... | 0.1 |
| ⋮ | | | ⋮ | | | ⋮ |

*Step 3: Final Feature Matrix*

**Figure 7.** Feature matrix.

## *3.3. BERT Framework*

In this section, we describe the overview and details of the BERT model. Bidirectional Encoder Representations from Transformers (BERT) is a pretrained language model based on the Transformer architecture. Its core idea is to learn rich language representations through large amounts of unsupervised pretraining data and then fine-tune the model on downstream tasks. Figure 8 shows the whole framework of BERT.

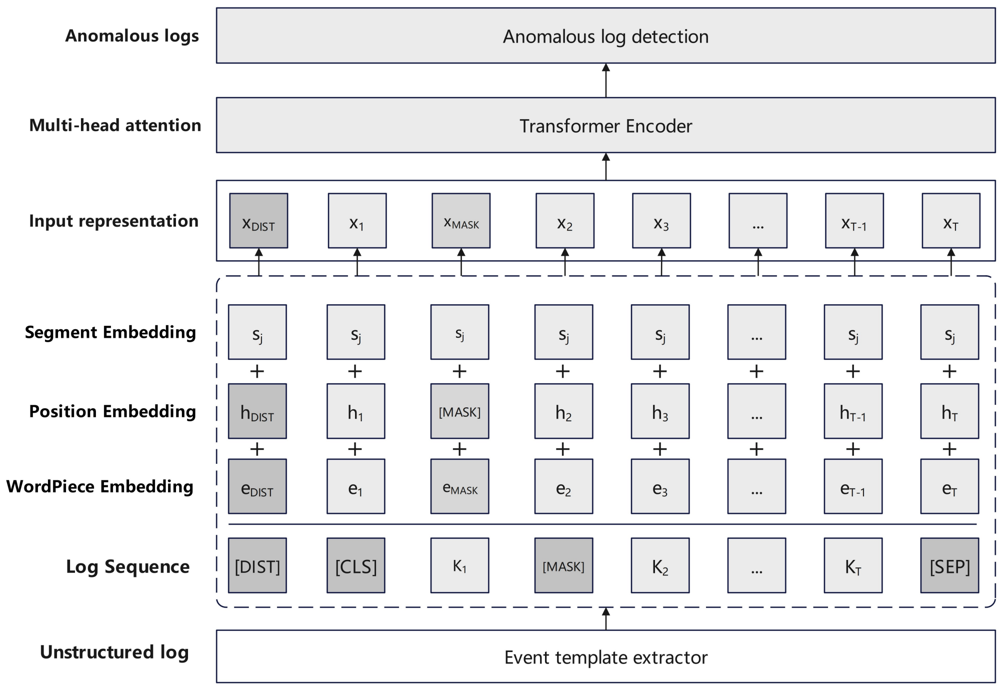


**Figure 8.** The logBERT framework.

### 3.3.1. Self-Attention Layer and Multi-Head Attention

In BERT, the input vector is formed by summing three different embeddings: WordPiece token embedding, segment embedding, and position embedding. Each computation of self-attention [18] involves three intermediate weight matrices: $W_q$, $W_k$, and $W_v$. They each linearly transform the input $x$ to generate the query, key, and value tensors.

Finally, self-attention can be defined as

$$Self-Attention(Q,K,V) = softmax\left(\frac{X^j \times W_Q \times W_K \times X^j}{\sqrt{d_k}}\right) \times X^j \times W_V \tag{4}$$

Multi-head attention uses parallel self-attention mechanisms to jointly capture different aspects of information for various log keys. Multi-head self-attention concatenates the outputs of multiple single-head self-attention layers into one and then passes it through a fully connected layer to reduce the dimensionality of the output. Multi-head attention can be defined as

$$f(X^j) = Concat(Attention_1, \ldots, Attention_H) \times W^O \tag{5}$$

Between layers, an Add&Norm layer is added, which employs residual connections and performs layer normalization. This mainly addresses the issue of vanishing gradients in neural networks. Finally, a feedforward network (FFN) is applied to deepen the network structure. The FFN layer consists of two linear operations with an ReLU activation function in between, corresponding to the following formula:

$$FFN(f(X^j)) = \max(0, f(X^j)W_1 + b_1)W_2 + b2 \tag{6}$$

### 3.3.2. Pretraining Tasks

**Masked Language Model.** The purpose of the MLM task is to enable the model to understand bidirectional context, thereby better capturing the semantic relationships within a log sequence. In the log masking process, we randomly selected 15% of the log keys for masking. These selected log keys were utilized as follows:

- Eighty percent were replaced with the [MASK] token.
- Ten percent were replaced with other randomly chosen log keys.
- The remaining 10% remain unchanged.

Let us assume that the input sequence is $x^j = \left(x_1^j, x_2^j, \ldots, x_N^j\right)$, where $x_i^j$ can be an original log key or a masked (MASK) token. The input sequence, after passing through the Transformer encoder, yields contextual embedding representations for each log key $H = (h_1, h_2, \ldots, h_N)$. For each masked log key$x_i^j$, its corresponding contextual embedding $h_i$ is used to predict its original log key through a softmax function:

$$\widehat{y_i} = Softmax(W_c h_i + b_c) \tag{7}$$

where $W_c$ and $b_c$ are trainable parameters during the training process and $\widehat{y_i}$ is the predicted probability distribution for the ith log key.

#### 3.3.3. Anomalous Log Sequence Detection

The idea of applying BERT to log anomaly detection is to derive an anomaly score for a log sequence based on the prediction results of the MASK tokens. To accomplish this, given a test log sequence, we first randomly replaced a certain proportion of log keys with MASK tokens and used this randomly masked log sequence as input for BERT. Then, we calculated the likelihood of log keys appearing in the positions of the MASK tokens. We constructed a candidate set containing the top g normal log keys with the highest likelihoods, as computed by $\widehat{y_i}$. If the actual log key was within this candidate set, then we considered it to be normal. Then, when a log sequence contained more than $r$ anomalous log keys, we labeled the entire log sequence as anomalous. Here, $r$ and $g$ are hyperparameters which need to be tuned based on a validation set.

## 4. Evaluation

In this section, we first introduce two datasets for testing: HDFS, and BGL. Next, we introduce the specific experimental settings and evaluation metrics and present the experimental results with discussions. We compare some state-of-the-art methods and present the following research questions (RQs):

- **RQ1:** How does the BERT framework perform in the log anomaly detection task compared to other traditional methods?
- **RQ2:** Does using TF-IDF for feature extraction improve the performance of the BERT framework?
- **RQ3:** Has the use of large language models improved the accuracy of log parsing? What impact does it have on the performance of anomaly detection in the framework?

### *4.1. Datasets and Metrics*

#### 4.1.1. Datasets

In this paper, we evaluate the performance of the best models on two datasets [19]: HDFS and BGL. A detailed introduction of the two datasets is shown in Table 4.

**Table 4.** Statistics of the datasets used for evaluating LogBERT.

| Dataset | Log Sequence No. | Log Key (Unique) No. | Anomalies |
|---|---|---|---|
| HDFS | 11,175,629 | 30 | 16,838 blocks |
| BGL | 4,747,963 | 267 | 348,469 logs |

**Hadoop Distributed File System (HDFS)**: The 11,175,629 log messages were collected from more than 200 Amazon EC2 nodes and formed the HDFS dataset. Each block operation, such as allocation and deletion, was recorded by a unique block ID. All of the logs were divided into 575,061 blocks according to the block ID and formed the log sequences. A total of 16,838 blocks were marked as anomalies. For HDFS, we grouped log keys into log sequences based on the session ID in each log message. The average length of the log sequences was 19.

**BlueGene/L (BGL) Supercomputer System**: The 4,747,963 log messages were collected from the BlueGene/L supercomputer system at Lawrence Livermore National Labs. There were 348,460 log messages labeled as anomalies. The BGL dataset has no unique ID for the log sequence, and thus a sliding window was used to obtain the sequence. The sliding window in this study consisted of the node ID, window size, and step size.

### 4.1.2. Baselines

We used the Drain and AEL parsers as baselines for unstructured log parsing tasks and compared them with the large language model ChatGPT. Subsequently, we employed the following five traditional methods as baselines for the log anomaly detection task to compare them with the BERT model:

(1) Log Parser Baselines

**Drain**: Drain is a log parsing algorithm designed to parse unstructured logs into structured formats by using a fixed-depth tree structure to capture log patterns efficiently.

**AEL**: Abstracting execution logs (AEL) is one of the state-of-the-art log parsing approaches which comprises four steps: anonymize, tokenize, categorize, and reconcile.

(2) Log Anomaly Detection Baselines

**Principal Component Analysis (PCA) [20]**: PCA analyzes log sequences by reducing dimensionality, identifying principal components, and detecting anomalies as deviations from normal sequence patterns. This method improves the detection of unusual behaviors and events in log data, enhancing anomaly detection efficiency and accuracy.

**Isolation Forest (iForest)**: Isolation forest efficiently detects log anomalies by isolating them in a binary tree. It partitions data, assigning anomalies to shorter paths and distinguishing them from normal instances.

**LogCluster [21]**: LogCluster efficiently detects anomalies by clustering log entries. It groups similar logs, identifying outliers as abnormal clusters.

**One-class SVM [22]**: One-class SVM applies a support vector machine to detect anomalies in log data. It constructs a hyperplane to separate normal instances from outliers in high-dimensional space.

**DeepLog [6]**: DeepLog utilizes deep learning to detect anomalies in logs. It employs recurrent neural networks to capture temporal dependencies and identify abnormal sequences.

**LogAnomaly [7]**: LogAnomaly detects anomalies by learning normal log patterns. It employs sequence modeling techniques like LSTM to capture temporal dependencies. Deviations from learned patterns are flagged as anomalies, enhancing detection accuracy.

### 4.1.3. Metrics

In order to evaluate the effectiveness of the proposed model in anomaly detection, the accuracy, precision, recall, and F1 score were used as evaluation metrics. What is more, to evaluate the performance of large language models on log parsers, the group accuracy, message-level accuracy, and edit distance were used. These metrics are defined as follows:

- Accuracy is the percentage of log sequences correctly detected by the model among all the log sequences.
- Precision is the percentage of anomalies correctly detected among all the detected anomalies by the model.
- Recall is the percentage of anomalies correctly detected by the model among all the anomalies.
- F1 score is the harmonic mean of the precision and recall. The maximum value of the F1 score is one, and the minimum value of the F1 s core is zero.

- Group accuracy measures the proportion of log messages correctly classified within predefined groups or clusters.
- Message-level accuracy evaluates the similarity between the predicted and actual log messages using the edit distance.
- The edit distance measures the minimum number of operations (insertions, deletions, and substitutions) required to transform one string into another. The edit distance can help evaluate how well a log entry matches multiple templates, thereby selecting the most suitable template for parsing the log.

## *4.2. Experiment Results*

### 4.2.1. Performance on Log Parser

Tables 5 and 6 use two datasets (HDFS and BGL, respectively) to evaluate three different log parsing methods (Drain, AEL, and ChatGPT). These methods were assessed for their performance in log anomaly detection tasks, with specific evaluation metrics including the group accuracy, message-level accuracy, and edit distance.

**Table 5.** Log parser results on HDFS dataset.

| **Method** | **HDFS** | | |
|---|---|---|---|
| | **Group Accuracy** | **Message-Level Accuracy** | **Edit Distance** |
| Drain | 0.9975 | 0.6255 | 0.9400 |
| AEL | 0.9975 | 0.6245 | 0.9425 |
| ChatGPT | 0.9610 | 0.9210 | 0.9265 |

**Table 6.** Log parser results on BGL dataset.

| **Method** | **BGL** | | |
|---|---|---|---|
| | **Group Accuracy** | **Message-Level Accuracy** | **Edit Distance** |
| Drain | 0.9625 | 0.3435 | 4.9725 |
| AEL | 0.9570 | 0.3435 | 5.0570 |
| ChatGPT | 0.8325 | 0.8700 | 3.9625 |

In the BGL dataset, Drain and AEL demonstrated exceptionally high group accuracy (approximately 0.96), indicating outstanding performance in effectively grouping logs. Their message-level accuracy was similar, suggesting no significant differences in classifying individual log messages. However, they exhibited a higher edit distance (approximately five), reflecting challenges in maintaining structural integrity when reconstructing log messages.

In contrast, ChatGPT showed lower group accuracy on the BGL dataset (approximately 0.83), possibly due to constraints in maintaining log structural integrity. Nevertheless, it displayed higher message-level accuracy (approximately 0.87) and a lower edit distance (approximately 3.96), highlighting its advantages in understanding and categorizing individual log messages.

On the HDFS dataset, Drain and AEL also demonstrated extremely high group accuracy (approaching 1.0) and lower edit distances, confirming their stability and effectiveness in log grouping and structural preservation. Comparatively, ChatGPT showed slight improvement on the HDFS dataset, with a slightly lower group accuracy (approximately 0.96) but higher message-level accuracy (approximately 0.92) and a lower edit distance (approximately 0.93), demonstrating efficiency in log semantic understanding and message-level classification.

In conclusion, deep learning methods like ChatGPT excel in understanding and classifying log messages, making them well suited for tasks requiring deep analysis and semantic understanding of log content. Therefore, for feature extraction in anomaly detection from logs, utilizing data inputs extracted by large language models can effectively capture the characteristic information of normal log sequence patterns.

### 4.2.2. Performance in Few-Shot Learning

Although large language models demonstrate impressive zero-shot capabilities, they still perform poorly in more complex tasks in a zero-shot setting. Few-shot prompting can be used as a technique to enable contextual learning. By providing log information and corresponding log templates in the prompts, we set the log parameters to meaningful words to guide the model in achieving better log parsing performance. As shown in Table 7, ChatGPT's parsing capabilities on the BGL dataset were evaluated in zero-shot, one-shot, and few-shot settings. With an increase in the number of sample prompts, the group accuracy significantly improved. However, the message-level accuracy remained at about 90%, except in the zero-shot setting. This indicates that large language models need sample prompts to learn good grouping and clustering effects for logs, but they can learn the internal template structure of each log well without samples.

**Table 7.** Different shot learning results on BGL dataset.

| **Method** | **BGL** | | |
|---|---|---|---|
| | **Group Accuracy** | **Message-Level Accuracy** | **Edit Distance** |
| zero shot | 0.1195 | 0.7260 | 9.8995 |
| 1 shot | 0.3395 | 0.9030 | 3.7420 |
| 2 shot | 0.6390 | 0.8590 | 3.2765 |
| 4 shot | 0.8325 | 0.8700 | 3.9625 |

### 4.2.3. Accuracy Evaluation of LogBERT and Baselines

Table 8 presents the performance comparison of LogBERT with six previous methods on both the HDFS and BGL datasets. Clearly, LogBERT outperformed all other methods, with an impressive F1 score of up to 90.8%. Count vector-based methods (such as PCA, LogCluster, and SVMs) achieved F1 scores ranging from 5% to 75% for anomaly classification. Semantic vector-based methods (such as DeepLog, LogAnomaly, and LogBERT) achieved F1 scores ranging from 60% to 90% for anomaly detection. This indicates that LSTM, Bi-LSTM, Word2Vec, and BERT are more suitable for capturing semantic information from log sequences.

**Table 8.** Performance comparison of different models on HDFS and BGL datasets.

| **Model** | **HDFS** | | | **BGL** | | |
|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1 Score** | **Precision** | **Recall** | **F1 Score** |
| PCA | 5.84 | 100 | 11.1 | 9.1 | 98.1 | 16.6 |
| LogCluster | 99.2 | 36.6 | 53.5 | 94.9 | 67.1 | 78.6 |
| One-Class SVM | 1.6 | 63.6 | 3.2 | 1.09 | 12.53 | 2.0 |
| Isolation Forest | 79.2 | 68.2 | 73.3 | 100 | 14.2 | 24.9 |
| DeepLog | 88.3 | 69.4 | 75.2 | 89.7 | 82.8 | 86.1 |
| LogAnomaly | 94.1 | 40.5 | 56.2 | 73.1 | 76.0 | 74.1 |
| Logbert | 87.1 | 78.2 | 82.3 | 89.4 | 92.3 | 90.8 |

Recall represents the percentage of anomalies correctly detected by the model among all anomalies. Prompt notification of anomalies is crucial for operations engineers. LogBERT achieved a recall of 92.3%, surpassing any other semantic-based method, with a 10% improvement over DeepLog and 16% improvement over LogAnomaly. This highlights the effective formation of semantic vectors through pretraining and preprocessing.

On the BGL dataset, the LogBERT model significantly outperformed the other models, demonstrating the benefits of anomaly classification models derived from sliding windows, pretraining, and large language model log parsing. The sliding window included the node ID, window size, and step size, enabling precise fault localization on each node for operations engineers. Pretraining and language model log parsing also provided more effective semantic information compared with the other methods.

### 4.2.4. Accuracy Evaluation of Anomaly Detection Using ChatGPT

Based on the data in Table 9, we conducted a comparative analysis of the performance of Drain, AEL, and ChatGPT in log anomaly detection tasks by using LogBERT on both the HDFS and BGL datasets.

**Table 9.** Performance comparison of Drain, AEL, and ChatGPT3.5 for LogBERT.

| **Method** | **HDFS** | | | **BGL** | | |
|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1 Score** | **Precision** | **Recall** | **F1 Score** |
| Drain + LogBERT | 87.1 | 78.2 | 82.3 | 89.4 | 92.3 | 90.8 |
| AEL + LogBERT | 86.5 | 79.4 | 81.2 | 88.1 | 91.4 | 89.3 |
| ChatGPT3.5 + LogBERT | 89.4 | 85.2 | 87.4 | 90.3 | 93.2 | 91.4 |

On the HDFS dataset, the performance of the three methods was relatively close, with ChatGPT slightly outperforming Drain and AEL. Specifically, ChatGPT achieved a precision of 89.4%, recall of 85.2%, and F1 score of 87.4%, which were slightly higher than Drain's 87.1%, 78.2%, and 82.3% and AEL's 86.5%, 79.4%, and 81.2%, respectively. This could be attributed to the smaller number of event templates in the HDFS dataset, making it easier to extract log semantics compared with BGL. Although ChatGPT slightly lagged behind Drain and AEL in terms of group accuracy, its advantage in semantic extraction was less pronounced on the HDFS dataset, thus resulting in less significant improvements in anomaly sequence recognition.

On the BGL dataset, the performance of the three methods was also similar, but ChatGPT once again demonstrated superior performance with a precision, recall, and F1 score of 90.3%, 93.2%, and 91.4%, respectively.

Overall, ChatGPT achieved optimal detection performance on both datasets, particularly excelling on the BGL dataset. This is likely because the BGL dataset had significantly more event templates (1848 compared with HDFS's 48) and a longer duration (214.7 days compared with HDFS's 38.7 h). Therefore, the relationships between the logs in the BGL dataset were more complex, posing challenges for traditional parsers in capturing semantic information. ChatGPT, as a large language model, excels in understanding and processing the semantics and patterns of log data, making it more accurate in identifying normal log sequences.

### 4.2.5. Comparison of Model Performance with the Latest Methods

To validate the novelty and effectiveness of our proposed method, we compared it with the latest methods on the BGL dataset. Table 10 shows the comparison of our method with LogFiT, MultiLog, LAnoBERT, and LogBERT in terms of precision, recall, and F1 score. Proposed by Lee et al. [23], LAnoBERT is a parser-free system log anomaly detection method using the BERT model, which learns a model through masked language modeling and performs unsupervised anomaly detection during testing using the masked language modeling loss function. Pham et al. proposed [24] MultiLog, which is a lightweight semi-supervised multi-task learning method, with key components including the pretrained BERT language model, dimensionality reduction, transformer attention mechanism, and multi-task learning. It does not use log parsers and adopts regular expressions to parse logs. Almodovar et al. proposed [25] LogFiT, which is a log anomaly detection model implemented as a Python package. LogFiT leverages a fine-tuned, pretrained BERT-based model to perform semantic analysis of logs without the need for intermediate parsing steps. LogFiT works directly on raw logs, eliminating the need for separate log parsing steps to extract log templates.

**Table 10.** A comparison of model performance with the latest methods.

| **Semi-Supervised** | **Log Parser** | **BGL** | | |
|---|---|---|---|---|
| | | **Precision** | **Recall** | **F1 Score** |
| LogFiT | X | 94.39 | 85.80 | 89.89 |
| MultiLog (Center) | X | 75.34 | 93.11 | 83.28 |
| LAnoBERT | X | - | - | 87.49 |
| LogBERT | O | 88.10 | 91.40 | 89.30 |
| LogBERT (LLMs) | O | 90.30 | 93.20 | 91.40 |

On the BGL dataset, the LogBERT method combined with large language models demonstrated significant superiority in log anomaly detection tasks. LogBERT (LLMs) harnesses the powerful semantic understanding capabilities of large language models, outperforming other methods in terms of precision, recall, and F1 score, achieving results of 90.30%, 93.20%, and 91.40%, respectively. Although it slightly underperformed relative to LogFiT's 94.39% precision, its higher recall and F1 score indicate that while LogFiT has an advantage in reducing false positives, it may be less effective in covering all anomaly events. This result suggests that LogBERT (LLMs) can more accurately capture anomaly patterns when processing complex log data, leading to better detection performance. In contrast, although other methods have their own strengths, especially in parser-free and lightweight design innovations, LogBERT (LLMs) still delivered the best overall performance, particularly showing higher precision and stronger coverage when handling complex log datasets.

## 5. Conclusions and Future Work

### 5.1. Conclusions

Log-based anomaly detection is crucial for helping operations personnel locate faults, improving operational efficiency, and enhancing the availability and reliability of large software systems. Our empirical research indicates that existing traditional log parsers suffer from inaccurate log parsing and are not effective in handling the semantic information of logs. To overcome the limitations of log parsing, this study proposes using the large language model ChatGPT-3.5 for log parsing. Additionally, this study introduced a BERT-based log anomaly detection model: LogBERT. By learning the patterns of normal log sequences, LogBERT is capable of identifying anomalous logs which deviate from these patterns. Our method uses few-shot learning to enhance the capability of a large language model in log parsing tasks. To better extract key information from structured logs, we employed sliding window event counting combined with TF-IDF weights to extract feature vectors. We evaluated the proposed method using two public datasets. The experiment results show that the structured logs obtained through large language model parsing were effective for log-based anomaly detection.

### 5.2. Future Work

Our research demonstrated the effectiveness of using large language models for log parsing in anomaly detection. However, there are still limitations to our experimental study. Our approach was based on learning the semantic meaning of log events. Given a log message, we first identified the template type of the log message while ignoring information such as timestamps, log security levels, and specific message content. However, in some cases, the ignored content may carry important information, such as specific log messages which might include node IDs, task IDs, IP addresses, exit codes, and timestamps, which may implicitly contain temporal anomaly information. This information could be useful for anomaly detection in certain situations. In our future work, we will encode more log-related information and investigate its impact on log-based anomaly detection.

Additionally, we identified the following major threats to effectiveness. (1) Subject datasets: In this work, we used datasets collected from distributed systems (i.e., HDFS) and supercomputers (i.e., BGL). Although these datasets come from real-world systems and contain millions of logs, the number of subject systems was still limited and did not cover all domains. In the future, we will evaluate the proposed method on more datasets collected from a variety of systems. (2) Noise in labels: Our experiments were based on two public datasets which are widely used in related works. These datasets were manually inspected and labeled by engineers. Therefore, data noise (false positives or negatives) may have been introduced during the manual labeling process. While we believe the amount of noise was minimal (if present), we will investigate data quality issues in our future work.

**Author Contributions:** Conceptualization, Y.Z. (Yihan Zhou) and Y.C.; methodology, Y.Z. (Yihan Zhou) and Y.C.; software, Y.Z. (Yihan Zhou); validation, Y.Z. (Yihan Zhou) and X.R.; formal analysis, Y.Z. (Yihan Zhou); investigation, Y.Z. (Yihan Zhou) and Y.Z. (Yukang Zhou); resources, Y.C.; data curation, Y.Z. (Yihan Zhou); writing—original draft preparation, Y.Z. (Yihan Zhou); writing—review and editing, Y.Z. (Yihan Zhou) and C.H.; visualization, Y.Z. (Yihan Zhou); supervision, C.H. and Y.L.; project administration, C.H.; funding acquisition, C.H. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was sponsored in part by the National Natural Science Foundation of China (No. 62177046 and 62477046), Hunan 14th Five-Year Plan Educational Science Research Project (No. XJK23AJD022 and XJK23AJD021), Hunan Social Science Foundation (No. 22YBA012), Hunan Provincial Key Research and Development Project (No. 2021SK2022), and High Performance Computing Center of Central South University.

**Data Availability Statement:** Data were derived from public domain resources. The data presented in this study are available in Loghub at https://github.com/logpai/loghub (accessed on 1 July 2024), reference number HDFS_v1/BGL. These data were derived from the following resources available in the public domain: 1. HDFS_v1: https://zenodo.org/records/8196385/files/HDFS_v1.zip?download=1 (accessed on 1 July 2024); 2. BGL: https://zenodo.org/records/8196385/files/BGL.zip?download=1 (accessed on 1 July 2024).

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

## References

1. Lin, X.; Lin, X.; Lagerstrom-Fife. In *Introductory Computer Forensics*; Springer: Berlin/Heidelberg, Germany, 2018.
2. Zhu, J.; He, S.; Liu, J.; He, P.; Xie, Q.; Zheng, Z.; Lyu, M.R. Tools and Benchmarks for Automated Log Parsing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 25–31 May 2019; pp. 121–130. [CrossRef]
3. Du, M.; Li, F. Spell: Streaming parsing of system event logs. In Proceedings of the 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 12–15 December 2016; pp. 859–864.
4. He, P.; Zhu, J.; Zheng, Z.; Lyu, M.R. Drain: An online log parsing approach with fixed depth tree. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; pp. 33–40.
5. Vinayakumar, R.; Soman, K.; Poornachandran, P. Long short-term memory based operation log anomaly detection. In Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, India, 13–16 September 2017; pp. 236–242.
6. 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, Dallas, TX, USA, 30 October–3 November 2017; pp. 1285–1298.
7. Meng, W.; Liu, Y.; Zhu, Y.; Zhang, S.; Pei, D.; Liu, Y.; Chen, Y.; Zhang, R.; Tao, S.; Sun, P.; et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In Proceedings of the IJCAI, Macao, China, 10–16 August 2019; Volume 19, pp. 4739–4745.
8. Zhang, X.; Xu, Y.; Lin, Q.; Qiao, B.; Zhang, H.; Dang, Y.; Xie, C.; Yang, X.; Cheng, Q.; Li, Z.; et al. Robust log-based anomaly detection on unstable log data. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 807–817.
9. Huo, Y.; Su, Y.; Lee, C.; Lyu, M.R. Semparser: A semantic parser for log analytics. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 881–893.
10. Yu, S.; He, P.; Chen, N.; Wu, Y. Brain: Log parsing with bidirectional parallel tree. *IEEE Trans. Serv. Comput.* **2023**, *16*, 3224–3237. [CrossRef]
11. Xu, J.; Cui, Z.; Zhao, Y.; Zhang, X.; He, S.; He, P.; Li, L.; Kang, Y.; Lin, Q.; Dang, Y.; et al. UniLog: Automatic Logging via LLM and In-Context Learning. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–12.

12. Xu, J.; Yang, R.; Huo, Y.; Zhang, C.; He, P. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–12.
13. Xiao, C.; Gou, Z.; Tai, W.; Zhang, K.; Zhou, F. Imputation-based time-series anomaly detection with conditional weight-incremental diffusion models. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Long Beach, CA, USA, 6–10 August 2023; pp. 2742–2751.
14. Xu, J.; Wu, H.; Wang, J.; Long, M. Anomaly transformer: Time series anomaly detection with association discrepancy. *arXiv* **2021**, arXiv:2110.02642.
15. Xu, W.; Huang, L.; Fox, A.; Patterson, D.; Jordan, M.I. Detecting large-scale system problems by mining console logs. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 117–132.
16. Sandhu, A.; Mohammed, S. Detecting Anomalies in Logs by Combining NLP features with Embedding or TF-IDF. *Authorea Prepr.* **2023**. [CrossRef]
17. He, S.; Zhu, J.; He, P.; Lyu, M.R. Experience report: System log analysis for anomaly detection. In Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada, 23–27 October 2016; pp. 207–218.
18. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1–11.
19. He, S.; Zhu, J.; He, P.; Lyu, M.R. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv* **2020**, arXiv:2008.06448.
20. Kurita, T. Principal component analysis (PCA). In *Computer Vision: A Reference Guide*; Springer: Cham, Switzerland, 2019; pp. 1–4.
21. Lin, Q.; Zhang, H.; Lou, J.G.; Zhang, Y.; Chen, X. Log clustering based problem identification for online service systems. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 102–111.
22. 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 (IEEE Cat. No. 03EX693), Xi'an, China, 5 November 2003; Volume 5, pp. 3077–3081.
23. Lee, Y.; Kim, J.; Kang, P. Lanobert: System log anomaly detection based on bert masked language model. *Appl. Soft Comput.* **2023**, *146*, 110689. [CrossRef]
24. Pham, T.A.; Lee, J.H. Lightweight Multi-task Learning Method for System Log Anomaly Detection. *IEEE Access* 2024, *early access*. [CrossRef]
25. Almodovar, C.; Sabrina, F.; Karimi, S.; Azad, S. LogFiT: Log anomaly detection using fine-tuned language models. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 1715–1723. [CrossRef]