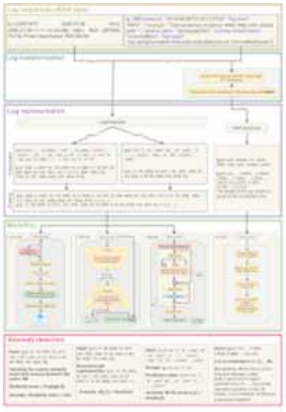
Anomaly Detection Algorithms for Real-Time Log Data Analysis at Scale

Publisher: IEEE Cite This PDF

András Horváth ; András Oláh ; Attila Pintér ; Bálint Siklósi ; Gergely Lukács ; István Z. Reguly All Authors

Abstract:
In recent years, Artificial Intelligence for IT Operations (AIOps) has gained popularity as a solution to various challenges in IT operations, particularly in anomaly detection. Although numerous studies have focused on detection, they often overlook cloud-based systems and the vast amount of data they generate. Moreover, the application of these results and managing multiple IT systems within diverse computing environments present notable challenges. In this paper, we explore anomaly detection in real-world environments, including legacy systems and cloud platforms, by evaluating existing methods and explicitly introducing four novel anomaly detection algorithms: Llama Model-based Log Anomaly Detection (LMLAD), Log Clustering with Cosine Similarity for Log Anomaly Detection (LCCLAD), Convolutional Auto-Encoder based Log Anomaly Detection (CAELAD), and Statistical Clustering for Log Anomaly Detection (SCLAD). These algorithms vary in complexity and accuracy, and offer tunable performance based on the specific requirements of the environment. In experiments on four log datasets (two private and two public), our best models improved F1-scores by 6–10% over state-of-the-art baselines, demonstrating the efficacy of our approach. Through empirical analysis, we demonstrate that while advanced techniques such as CAELAD often deliver high accuracy, simpler methods such as SCLAD can be equally effective in practical settings, depending on the complexity of the problem. Our results underscore the importance of selecting the right balance between sophistication and simplicity, challenging the assumption that the most sophisticated methods are necessary for effective anomaly detection in real-world log data.



Our proposed anomaly detection workflow consists of five steps, with the initial three consistent across all evaluated methods. The final two steps highlight our four dif... Show More

Published in: IEEE Access (Volume: 13)

Page(s): 136288 - 136311

DOI: 10.1109/ACCESS.2025.3594469

Date of Publication: 31 July 2025

Publisher: IEEE

Electronic ISSN: 2169-3536

Funding Agency:

documentation.

SECTION I.
Introduction

The rise of digitalization has profoundly transformed public administration, replacing traditional paper-based processes with online administration, commonly known as e-government. This digital shift introduces new challenges, including the need to detect unusual system behavior or usage patterns, a task commonly referred to as anomaly detection. A major challenge is that such anomalies form an open-ended set, making their identification especially complex. Although the application of state-of-the-art methods has been explored in the literature, they are often evaluated on benchmark datasets that are either outdated or syntetically generated. Consequently, they usually fail to represent real-world companies' current anomaly detection challenges. This shift poses new challenges for engineers working in IdomSoft Zrt.1, a 100% indirectly state-owned company that plays a leading role in the development, integration, installation, and operation of complex IT systems.

The systems operated by IdomSoft Zrt. are heterogeneous, comprising both legacy technologies and modern cloud-based solutions, using Platform as a Service (PaaS), for example, the Red Hat OpenShift Kubernetes [1] distribution. Due to the age of the systems and technologies they use, various types of data are generated during their operation. Specifically, legacy systems generate unstructured or semi-structured text data, including logs such as application, access, system, and audit logs, along with metrics such as CPU usage, memory consumption, network activity, etc. In contrast, modern cloud-based applications produce telemetry data (logs, traces, and metrics) from various service layers. In this paper, we focus on log analysis considering the unique property (diversity in structure, high volume and velocity, contextual information, etc.) of the data. Specifically, metrics and traces are structured data formats, whereas logs typically consist of semi-structured JavaScript Object Notation (JSON) generated by modern logging frameworks.

Companies operating IT systems, such as IdomSoft Zrt., produce a substantial amount of log data during operation. It is essential to store and analyze this data efficiently to optimize system performance and troubleshooting, help decision-making, minimize operation costs, and reduce security risk by identifying vulnerabilities and maximizing the reliability and availability of the systems. To meet these requirements, Artificial Intelligence for IT operations (AIOps) [2], [3] should be introduced and further optimized using advanced deep learning algorithms and statistical models. In the market, various production-ready anomaly detection algorithms have been integrated into observability platforms, such as OpenSearch [4], Splunk [5], Dynatrace [6], and more. However, these algorithms have limited capabilities; for example, they cannot be customized or optimized to meet specific user requirements, including the use of more advanced techniques such as neural networks. Over the past few years, several studies have been conducted in this area, focusing on log-based anomaly detection, such as DeepLog [7], LogAnomaly [8], LogRobust [9], and LogBert [10], etc. as well as on cybersecurity solutions that utilize big data and cloud infrastructure for real-time data processing [32]. Due to the unstructured nature of the data, its format and semantics can vary significantly between different systems. In most cases, log parsing is required for data analysis because it allows meaningful interpretation. Supervised learning models demand labeled data, which require domain expertise and can be costly because of the number of systems involved. However, unsupervised learning might produce incorrect results or false alarms, thereby reducing its reliability and precision. To simplify the log analysis, a unified log format was introduced at IdomSoft Zrt., which standardizes the data format across sources, simplifies data correlation, improves interoperability, and facilitates pre-built dashboards and queries. In addition, we introduce novel anomaly detection methods that can be applied to both unstructured and semi-structured (JSON) data formats. These methods eliminate the need for log parsing and data labeling and can be utilized separately or in a combined manner, depending on the complexity of the problem.

The main contributions of this paper are as follows. First, we present existing log anomaly detection methods, detailing all the steps involved in log analysis in a unified manner. This provides a foundational understanding of current methodologies in the field. Building on this, we introduce four novel anomaly detection methods, each offering unique approaches to enhance the detection process: Llama Model-based Log Anomaly Detection (LMLAD), Log Clustering with Cosine Similarity for Log Anomaly Detection (LCCLAD), Convolutional Auto-Encoder based Log Anomaly Detection (CAELAD), and Statistical Clustering for Log Anomaly Detection (SCLAD). To evaluate the effectiveness of these methods, we conducted a comprehensive analysis using private and publicly available datasets. Our evaluation considered critical factors such as computational complexity and operational costs. We observed that in practical applications, lower-complexity methods can achieve comparable performance to state-of-the-art, high-complexity approaches. This observation underscores the importance of balancing the accuracy and computational efficiency when selecting an optimal method for log anomaly detection. Moreover, our results reveal a discrepancy between benchmark datasets and real-world data, while absolute F1 scores differ, the relative ranking of methods remains consistent. This suggests that, although benchmarks may not reflect accurate real-world performance, they still offer a reliable basis for comparing the effectiveness of different approaches.

The remainder of this paper is structured as follows: Section II presents an overview of related log analysis techniques, focusing on relevant anomaly detection algorithms. In addition, it offers insights into the current trends in log analysis and highlights the gap between academic research and industry practices. In Section III, the proposed anomaly detection methods are introduced, along with a discussion of their strengths and weaknesses. In Section IV, we evaluate existing baseline methods and our proposed anomaly detection approaches using private and publicly available datasets. In Section V, we perform a comparative analysis considering the algorithms' performance, complexity, and operational costs. Section VI then presents the total cost of ownership associated with deploying these algorithms in a production environment. Section VII concludes the study by synthesizing the key findings and outlining potential directions for future research.

SECTION II.
Motivation and Related Work

Telemetry data, including logs, play a crucial role in IT operations. Logs provide detailed information about specific events within a complex system and are the main pillars of Root Cause Analysis (RCA) and anomaly detection. The logs are free-text messages developers write to describe a system's state or business logic. In contrast, various predefined log formats are available to ensure uniform or semi-structured data in which the log message is stored. One of the most commonly used and well-known log formats is the Syslog format in Unix-based operating systems or Windows event logs. However, there are many other formats such as JSON log format, Graylog extended log format, Common log format (web server access logs such as Apache access log, etc.), and W3C extended log format. In the past, it was common for application logs to be unstructured. Consequently, the most widely used method involves the use of rules for log parsing, such as regular expressions, machine learning, and other statistical-based methods. However, this rule-based approach relies on manually crafted rules provided by domain experts that vary for each application. Additionally, with the application architecture's evolution, logging has changed significantly.

- Monolithic architecture: The application consists of different layers that are usually logged into text files. These applications can be scaled vertically, meaning multiple instances cannot be run simultaneously on different hosts. Hence, each component logs into a single file that is stored on the same server. It is uncommon to have a dedicated solution for monitoring systems.

- Service-oriented architecture (SOA): It focuses on discrete services instead of a single monolithic service. The system's functionalities are exposed through a service bus, and each functionality typically has a dedicated component/service. Central logging must be used in most cases if we consider the number of components. Application Performance Monitoring (APM) has also become crucial in such systems.

- Microservice architecture: It has transformed application monitoring, including logging as well. Services are divided into interconnected, individual microservices. This change introduces the concept of Observability, which relies on telemetry data. In a microservice architecture, distributed tracing has become essential. A centralized logging system is crucial, as services typically run in the cloud, where operators often do not have direct access to the underlying infrastructure.

- Serverless architecture: This allows developers to run services without managing the underlying infrastructure. The service is built upon functions, each implementing a small fraction of the business logic. These functions are atomic parts of the services, and can be scaled up and down easily. A centralized solution is commonly used for logging purposes.

To impose the structure of the telemetry data semantic convention, which prescribes how certain common dimensions should be named in telemetry and which values can be assigned to them, would be defined. One such example is the Elastic Common Schema (ECS) [15], an open-source specification that defines a standard set of fields for storing event data, such as logs, traces, and metrics, in a JSON semi-structured format. In recent years, OpenTelemetry [16] has played a significant role in this area. It became the open Observability standard and is supported by most of the observability platforms. In such cases, the log format is JSON documents, significantly simplifying data processing. Shkuro et al. [17] introduced the importance of telemetry metadata, which is essential for schema-first applications. Telemetry-based Observability has emerged as a go-to solution for achieving visibility in modern distributed architectures and applications, becoming the industry's de facto standard. However, most of the research did not consider these effects. In [19] paper, academic and Microsoft researchers highlighted the gap between industry and research.

In recent years, research has moved significantly towards log parsing-based log analysis [7], [8], [9], [11] using deep learning and traditional machine learning techniques, which often involve incorporating a log parsing stage at the beginning of the log analysis pipeline. The Transformer architecture [12] has become increasingly popular and is frequently utilized for log analysis tasks, including the use of Large Language

Models (LLMs) to generate vector embedding [9], [10], [11], [14] and prompt engineering techniques [13]. Existing log analysis techniques presented in various research papers do not consider the latest trends in IT operations. To address the gap between academia and industry, we propose new methods suited to traditional and modern log analysis challenges and can be utilized in pipeline processing of real-world semi-structured (JSON) company data. The following steps are commonly used in log analysis within an industry:

- Log collection: Gathering logs from various systems, services, and applications.
- Log transformation: Converting unstructured logs into a structured format for easier processing and analysis or for further processing of structured logs, such as sampling, filtering, and tagging.
- Log storage: The transformed logs are stored in a centralized location, such as a big data platform or a data lake, to ensure efficient retrieval and long-term retention.
- Log analysis: Processing structured logs to extract meaningful insights, patterns, or anomalies that can help diagnose problems or optimize performance.
- Visualization and reporting: Presenting the analyzed data using dashboards, charts, or reports to facilitate understanding and decision-making.
- Alerting: Setting up automated alerts for critical events or thresholds to enable proactive monitoring and quick response to issues.

This paper focuses on the log analysis workflow as shown in Figure 1 without considering the log collection and centralized storage. A typical workflow for unstructured log data begins with a parsing phase, whereas this step is omitted for semi-structured data. This is followed by a log transformation, which may involve various techniques to harmonize the log data further. The second phase of the workflow is a log message representation that involves creating a mathematical representation of log sequences. The most crucial phase of the workflow is modeling. Supervised learning is uncommon in log analysis because millions of logs need to be labeled. As a result, unsupervised and self-supervised algorithms are the most widely used approaches. The final phase of the workflow is anomaly detection, which involves setting thresholds and determining the log sequences that are classified as anomalies.

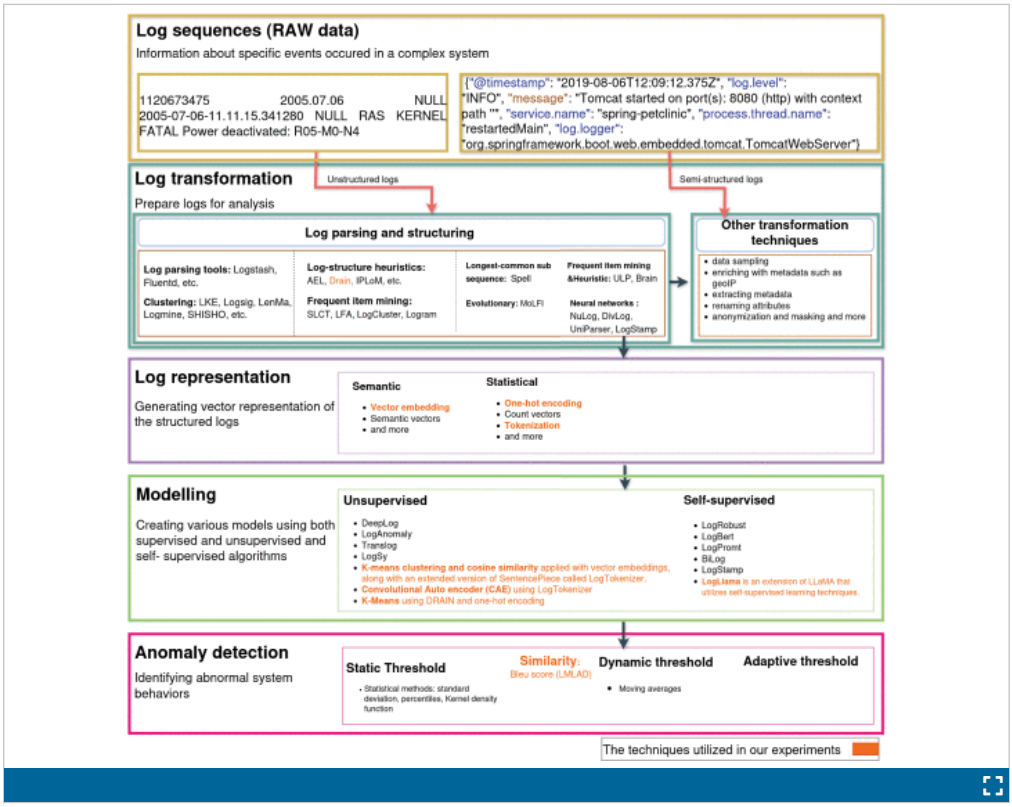


FIGURE 1.
Overview of commonly applied anomaly detection workflows. Most approaches have five distinguishable steps, applying various methods at each stage. The methods highlighted in orange represent the focus of our study.

SECTION III.
Log Anomaly Detection Methods

In this section, we review the current methods for log anomaly detection and their strengths and limitations. Then, we present our anomaly detection approaches.

A. Available Methods

To evaluate the performance of our anomaly detection methods, we selected three open-source techniques as the baseline:

- LogBert [10] is a self-supervised framework designed to identify log anomalies. It leverages Bidirectional Encoder Representations from Transformers (BERT) to understand the typical patterns of normal log sequences through two novel self-supervised training tasks. Consequently, LogBert can detect log abnormalities where deviations occur from these normal log patterns.

- DeepLog [7] is an anomaly detection technique that uses Long Short-Term Memory (LSTM) to model a natural language sequence. DeepLog automatically learns log patterns from normal execution and detects anomalies when log patterns diverge from the model built on logs generated during normal operations.

- LogAnomaly [8] is based on the LSTM network that automatically detects sequential and quantitative anomalies simultaneously. LogAnomaly was inspired by word embedding using Template2Vec, to extract semantic and syntax information from the log templates accurately.

The baseline methods outlined above depend on Drain [30] for log parsing, which can become computationally intensive when processing high volumes of log messages (for example, in normal operation processing 30,000 log entries per second or, in some cases, over 100,000 log entries per second). LogAnomaly utilizes Template2Vec, which is based on Word2Vec. DeepLog converts the Drain output into a sequence of log keys, whereas LogBERT employs a transformer encoder to generate embedding vectors for log keys. DeepLog and LogAnomaly rely on LSTM-based modeling, which is replaced by a transformer-based architecture in LogBERT to capture the long-term dependencies in sequences more efficiently. However, the requirement for log parsing persists. Most existing anomaly detection methods utilize sliding, fixed, time-based, or session-based windows that group normal and anomalous log messages within events. Depending on the size of the window and volume of the data, an anomalous event may contain more than 1,000 log messages. As a result, identifying the exact log message (or error) that causes the issue becomes difficult. Consequently, one of the main focuses of this research is anomaly detection at the event (log) level, which is addressed and presented in detail.

B. Proposed Methods

Considering the limitations of existing anomaly detection methods discussed in the previous sections and the size and complexity of logs, we developed four innovative anomaly detection methods designed to address these challenges and improve detection performance. The overall pipeline of the proposed anomaly detection methods is illustrated in Figure 2, considering the anomaly detection workflow illustrated in Figure 1. Among the proposed methods, only SCLAD utilizes Drain for log parsing. Other methods employ our novel LogTokenizer to generate vector representations of logs directly from raw log sequences. In addition to the LogTokenizer, LCCLAD also uses vector embedding through OpenAI service. The four proposed methods span a spectrum of complexity: three leveraging large-language-model techniques (LMLAD, LCCLAD, CAELAD) and one purely statistical method (SCLAD) for low-resource settings.

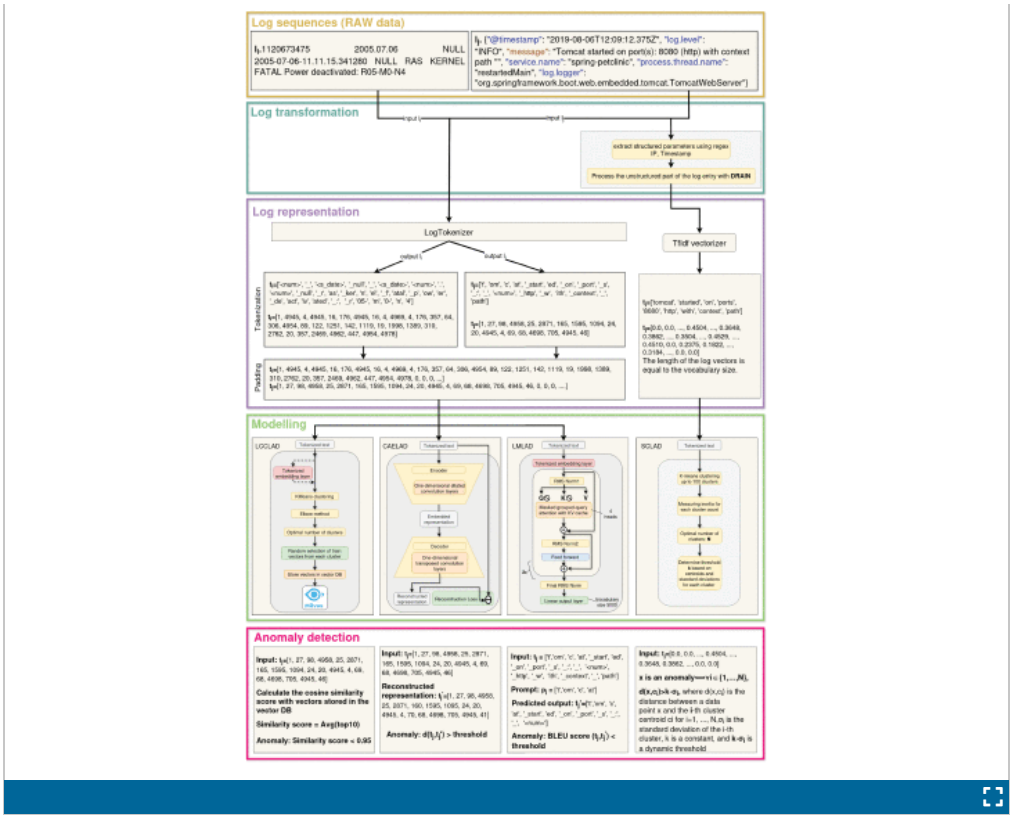


FIGURE 2.
This figure illustrates the algorithmic steps of commonly used anomaly detection workflows (extracted from Figure 1), describing the same five major steps, but in more detail and with a focus on the algorithmic solutions of this study. The first three steps are identical across all investigated methods, while the last two steps depict our four different approaches.

LogTokenizer: We developed LogTokenizer, an extension of the official Llama tokenizer, which is based on the language-independent SentencePiece tokenizer [18] and specifically customized to process log messages. Inspired by the capabilities of masked language models, we introduced specific entities within logs that could effectively serve as "MASK" tokens. To better capture the semantics of log messages, we incorporated unique tokens to represent key entities. These tokens include:

<token_with_digit>, <email_address>,

<ip>, <duration>, <path>, <url>,

<utc_date>, <klog_date>, <custom_date>,

<mask>, <cls>, <s_date>, <empty_list>,

<empty_set>

These tokens enhance the efficient mapping of specific entities within log messages, improving their structured representation. A comparison between Llama tokenizer and LogTokenizer is presented in Appendix B. The LogTokenizer was trained on 20 million log lines, consisting of both Hungarian and English log messages, with Hungarian accounting 61% and English for 39% of log data. The configuration included the use of Byte-Pair Encoding (BPE) as the model type, a maximum sentence length of 512 tokens, and a vocabulary size limited to 5,000.

Let l represent a single log line, and let V denote the vocabulary of the LogTokenizer with a fixed size of $|V| = 5000$.

The LogTokenizer maps log line l to a sequence of tokens $t_1, t_2, \ldots, t_m$, where each token $t_j \in V$ for $j \in \{1, 2, \ldots, m\}$, and m is the total number of tokens in the log line.

This can be expressed as:

$$l \xrightarrow{\text{Tokenizer}} \{t_1, t_2, \ldots, t_m\}, \quad t_j \in V, \quad |V| = 5000.$$

View Source

Each log line l can contain a variable number of tokens, resulting in sequences of different lengths. To ensure uniformity, we applied padding to normalize the length of each tokenized sequence $t_1, t_2, \ldots, t_m$ to a fixed length L:

$$\{t_1, t_2, \ldots, t_m\} \xrightarrow{\text{Padding}} \{t_1, t_2, \ldots, t_m, \underbrace{0, \ldots, 0}_{L-m}\},$$

View Source

where 0 is the padding token.

1) Llama Model-Based Log Anomaly Detection (LMLAD)

In log analysis, the semantic and contextual meaning of log messages is crucial. Consequently, modern log analysis frameworks, such as LogRobust, integrate semantic and contextual information to enhance anomaly detection [9]. LogRobust utilizes an attention-based Bi-LSTM neural network capable of capturing the contextual dependencies within log sequences and assigning varying importance to different log events [9]. This design enables LogRobust to effectively manage unstable and variable log sequences. However, as a supervised learning model, LogRobust requires labeled data sets containing normal and anomalous logs for training, which entails substantial manual annotation efforts. In addition, its performance can be significantly degraded by noise arising from mislabeled logs.

The transformer model introduced in [12] uses the attention mechanism to better understand the context and relationships between words in a sentence or document. Transformer-based models have demonstrated significant potential in anomaly detection, particularly due to their ability to operate without requiring labeled data. As discussed in Section III, LogBERT—a Transformer-based framework—achieves strong performance in this area. However, LogBERT relies on log parsing techniques such as Drain or other parsers as a preprocessing step, which adds additional complexity and computational overhead. Other transformer-based methods, such as TranConvAD [33], do not require traditional log parsing, but they still involve log processing to extract information like timestamps, log levels (treated as risk types), and semantic content. A key limitation of TranConvAD is its need to extract and separately process multi-dimensional features during preprocessing, which incurs significant time costs. Moreover, its dependence on a large set of labeled log sequences reduces its adaptability to newly deployed systems. NeuralLog [34] completely eliminates the need for log parsing by leveraging BERT, a widely used pre-trained language model, to capture the semantic meaning and contextual information of raw log messages. Then, it applies a transformer-based classification model to anomaly detection. In the paper by Diego et al. [35], the researchers introduced MAIA, an LLM-based approach that is a strong contender for cloud-native systems. It also utilizes a dynamic Bayesian network to generate probabilistic inferences about incidents. MAIA involves fine-tuning LLaMA 3 8B on normal execution logs, which is computationally intensive and still requires refinement to reduce false positives and mitigate alert fatigue. In contrast, the proposed LMLAD leverages the capabilities of large language models to capture long-range dependencies and complex relationships within log sequences, without requiring log preprocessing or parsing. This eliminates the need for a preprocessing stage, simplifying the analysis pipeline. Furthermore, LMLAD benefits from advances provided by the LLAMA model, including enhanced generative capabilities due to the autoregressive architecture and improved positional encoding through Rotary Positional Embedding (RoPE), leading to a more flexible and powerful approach to log-based anomaly detection. Furthermore, LMLAD has significantly fewer parameters (19 million) compared to MAIA (8 billion) and other transformer-based models presented above (BERT-Base 110 million and BERT-Large 340 million). Despite its smaller size, it still benefits from the Hugging Face ecosystem, offering quantization and model serving features. Additionally, since LMLAD is trained from scratch, it eliminates the need for fine-tuning, streamlining the log anomaly detection process by removing an extra step.

Current state-of-the-art LLM models are not tailored to the analysis of specific log data. To address this limitation along with those previously discussed, we developed LMLAD - an optimization of the Llama model [20] - specifically designed for analyzing log data.

LMLAD is a transformer decoder model that utilizes an auto-regressive approach, and its architecture aligns with the Llama design but incorporates slight modifications outlined in Tables 1 and 2. The model comprises 19.6 million parameters, significantly decreasing training and operational expenses without compromising performance.

TABLE 1 LMLAD Specific Hyperparameters

#vocab	#dim	#layers	#att. heads	#embeddings
5000	1024	2	2	1024

This approach does not require domain-specific knowledge of a particular field. Instead, we should choose a time window when a specific system functions correctly and utilize these logs for training. Test system logs can even be used instead of production-specific logs. For training, we utilized the Trainer module from Hugging Face Transformers library, with the hyperparameters outlined in Table 2.We employ a one-shot prompt engineering approach to evaluate each log message and calculate the BLEU score [21]. The prompt for a given log message *l* is created using Algorithm 1 and is provided to the LMLAD model.

TABLE 2 LMLAD Specific Training Parameters

#epoch	learning_rate	weight_decay	#batch
5	1e-5	0.01	512

Algorithm 1 Generates Prompts From Logs
Require: A list of *logs*, end-of-sequence token *eos_token*.

Ensure: List of generated *prompts*.

1: Initialize empty list *prompts*.

2: Define the *splitter* as regex: r'([|:|(|)|[|]|'|||"|,|=])'.

3: **for** each string *log* in *logs* **do**

4: Split *log* into tokens using the *splitter* and store in *tokens*.

5: Initialize *weighted_prompt* to 2.

6: **if** length$(tokens) \geq 10$ **then**

7: Increment *weighted_prompt* by $\lfloor 0.1 \times \text{length}(tokens) \rfloor$.

8: **end if**

9: Create *log_line* by concatenating *i*, *eos_token*, and the first *weighted_prompt* tokens from the *tokens*.

10: Append *log_line* to *prompts*.

11: **end for**

12: **return** *prompts*.

Let $\mathbf{x} = \text{LogTokenizer}(p)$ represent the tokenized prompt (sequence of tokens), where:

$$\mathbf{x} = \{x_1, x_2, \ldots, x_m\},$$

View Source

where m denotes the number of tokens, and p is the prompt generated by Algorithm 1.

The LMLAD model generates a sequence of tokens $\mathbf{y} = \{y_1, y_2, \ldots, y_k\}$, where $k \leq 50$ (number of new tokens). Each token y_t is sampled from the model probability distribution as follows:

$$P(y_t = w_i \mid X_{<t}) = \frac{e^{z^{t,i}/T}}{\sum_{j=1}^{V} e^{z^{t,j}/T}},$$

View Source

where:

- $z^{t,i}$ is the i-th logit value produced by the model for time step t,
- V is the vocabulary size,
- w_i is a specific token
- $T = 1$ is the temperature parameter used to control the randomness of sampling.

Because num_beams $= 1$, the next token y_i is selected deterministically by considering the token with the highest probability:

$$y_t = \arg\max_y P(y_t = w_i \mid X_{<t}).$$

View Source

The generation process stops when one of the following conditions is met:

- $i = k$, where $k = 50$ (maximum tokens generated).
- $y_i =$ pad_token_id (PAD token is generated, indicating the end of the generation).

The final output sequence $\mathbf{y} = \{y_1, y_2, \ldots, y_k\}$ is concatenated with the original tokens $\mathbf{x}$ to form the complete log message:

$$\mathbf{X}_{\text{output}} = \mathbf{x} \oplus \mathbf{y}.$$

View Source

The final step is to calculate the BLEU score using the generated X_{output} and the original log message l. A low BLEU score for a specific log message suggests that this log was not present during regular operations, indicating an anomaly. A threshold can be defined based on the BLEU score deviation from the calculated average or by using the distribution plot of the BLEU scores.

2) Log Clustering with Cosine Similarity for Log Anomaly Detection (LCCLAD)

The LCCLAD method is a fully unsupervised anomaly detection approach that clusters semantically similar log messages in vector space and identifies anomalies as outliers to these clusters. It is optimized for lightweight, real-time deployment in production environments with minimal computational overhead. This setup is conceptually similar to retrieval-augmented generation (RAG), where semantic similarity is used to retrieve relevant information based on vectorized queries. The proposed method works similarly to that introduced by Pan et al. in their preprint work [26].

Log messages are embedded into dense vectors using either OpenAI's text-embedding-3-small model [28] or the lightweight LogTokenizer. The OpenAI model outputs fixed 256-dimensional vectors and was used only on public datasets to benchmark the semantic performance of the LogTokenizer. The LogTokenizer generates variable-length vectors, which are padded to match the longest log entry within each dataset. All vectors are normalized to enable cosine similarity computations.

For efficient approximate nearest neighbor (ANN) search, vectors are stored in Milvus [27], a high-performance open-source vector database using an IVF_FLAT index with $nlist = 128$. The system operates in a Kubernetes-based architecture with CPU-only containers. No metadata filtering is applied; similarity comparisons rely solely on log content. This enables cross-domain generalization while maintaining data locality and compliance with data protection requirements.

K-Means clustering is used to partition the historical log vectors, with the number of clusters selected via the elbow method and optionally validated by the silhouette score. From each cluster, a fixed number of representative vectors presumed to reflect normal system behavior are randomly sampled to form the training set. Importantly, we do not use cluster centroids for modeling; instead, actual log vectors are selected to preserve semantic granularity. Clusters dominated by anomalies contribute proportionally fewer vectors.

The inference phase compares incoming log vectors to the training set using cosine similarity. For each test vector, the similarity scores of its ten nearest neighbors in the training set are averaged, and the result is compared to a dataset-specific threshold (denoted as τ). If this mean similarity falls below τ , the log entry is

flagged as anomalous. Threshold values τ were empirically determined for each dataset based on validation experiments, typically ranging between 0.9 and 0.98, to favor high recall while maintaining acceptable precision.

LCCLAD does not incorporate metadata such as log source, component, or timestamp into the anomaly detection process. All similarity comparisons are based solely on the content of the log message. This allows cross-domain anomaly detection but may lead to occasional false positives when unrelated logs have similar textual structure.

This fully unsupervised, content-based clustering method is particularly effective in detecting novel anomalies without requiring labeled training data or retraining. It is well suited for environments where semantic anomalies are more critical than structural consistency.

The LogTokenizer embedding method was ultimately chosen for deployment due to its low computational cost, fast inference, and ability to run entirely in-house without sharing sensitive data externally. The most computationally expensive operations—-K-Means clustering and index building—-are only performed during initial setup or after substantial changes in log structure.

3) Convolutional Auto-Encoder Based Log Anomaly Detection (CAELAD)

Autoencoders are frequently employed for anomaly detection because they require only non-anomalous data for training. Due to the compressed latent representation between the encoder and decoder, the neural network learns to capture the typical patterns in a lower-dimensional manifold. When new data deviate from the samples presented during training, their reconstruction is less accurate, and this discrepancy can be used to identify anomalies. By adjusting the threshold of the reconstruction loss, this method allows for optimization of the recall and precision of the architecture.

In our investigation, we implemented a simple convolutional autoencoder with three convolutional layers in the encoder, consisting of 16, 32, and 64 channels, each with a stride of two, reducing the data dimensions at every layer. In the decoder, transposed convolutions were used to upscale the data in a similar manner. The inputs to our model were the token vectors produced by our LogTokenizer, which were introduced earlier. We chose convolutional autoencoders over LSTM or fully-connected structures to capture local token patterns while keeping the computational complexity on an acceptable level.

We set the reconstruction loss threshold to maximize recall and find as many anomalies as possible, even if false positive samples occur.

4) Statistical Clustering for Log Anomaly Detection (SCLAD)

The proposed anomaly detection method combines statistical techniques with machine learning. The initial step involves pre-processing log entries using the Drain algorithm, which extracts structured templates and variable components from raw log data. The Drain algorithm is an online log parsing method that can process log entries in a streaming manner. It utilizes a fixed-depth parse tree to efficiently extract structured templates and variable components from the raw log data. The resulting templates were tokenized based on whitespace and selected special characters. The tokens were vectorized using Term Frequency-Inverse Document Frequency (Tf-Idf) transformation, ensuring that the vectors reflect the distribution of words within the log entries.

The importance of term t in a document d within a corpus D is calculated as:

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

View Source

where

$$\text{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

View Source

and

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

View Source

In these equations:

- $f_{t,d}$: the frequency of term t in document d
- N: the total number of documents in the corpus
- $|\{d \in D : t \in d\}|$: the number of documents containing term t

The Tf-Idf formula quantifies the importance of a term in a document in a corpus, balancing its frequency in a specific document with its occurrence across the entire corpus.

Subsequent to vectorization, these feature vectors were used to train the K-Means clustering algorithm. The optimal number of clusters was determined using the elbow method [31], in which the inertia values were plotted as a function of the number of clusters. The point with the most significant change in inertia was selected as the optimal number of clusters.

Once the K-Means model is trained using the number of clusters determined by the elbow method, it is applied to anomaly detection in the following manner: for the training data, which does not contain anomalous data, we compute the centroids and standard deviations for each cluster. During the testing phase, data points whose distance from the centroid of any cluster exceeded a threshold of k times the standard deviation were classified as anomalous.

The parameter k optimization follows a systematic approach designed for operational deployment and computational efficiency constraints. SCLAD is intentionally designed as a resource-efficient pre-filtering method that can identify potentially suspicious log entries for subsequent analysis by more computationally intensive methods. A lower k value leads to a larger number of log entries being flagged as anomalies by the algorithm. For k value selection, we again applied the elbow method on the training data, which contains no anomalies. With increasing k values, more data points are classified as belonging to their respective clusters, resulting in fewer samples being flagged as anomalies. We selected the k value at the point where this decreasing trend exhibits a sharp change in slope, following the elbow method principle.

The parameter k is adjustable and controls the sensitivity of the detection system with the understanding that this first-stage filtering trades some detection capability for significant computational savings and operational feasibility. Having designed this method for pre-filtering purposes, we now focus solely on evaluating its performance characteristics.

SECTION IV.
Experiments

To evaluate our methods, we conducted experiments using two publicly available dataset from LogHub [22] and two private datasets, as summarized in Table 3. The logs of the private datasets are stored on the IdomSoft Observability Platform [24], which is built on OpenSearch.

TABLE 3 Dataset Summary Information with Anomaly, Train and Test Log Distribution

Dataset	Messages	Unique Messages	Unique Templates	Anomaly	Train	Test	
						Normal	Anomaly
BGL	4,747,963	358,352	777	348,460	2,676,916	20,000	20,000
Thunderbird	186,474,685	23,028,776	5420	161,284	2,843,286	20,000	20,000
VReg	11,602,971	52,366	295	38,643	914,537	10,000	10,000
ICard	452,847	29,504	77	85,974	168,045	10,000	10,000

Figure 3 illustrates the real-world log collection pipeline, which is a component of the IdomSoft Observability platform. Logging requirement of IdomSoft Zrt. is that application logs should be in the JSON format and comply with the ECS. These logs should be generated using any log framework, such as Log4j, and sent to the IdomSoft Observability Platform centralized log management system via the Fluentd [29] data collector. Because the log files are already in a semi-structured JSON format, we focus on creating vector representations of the log messages, anomaly detection algorithms, and operation cost and efficiency. We developed a custom-built framework to retrieve logs from the IdomSoft Observability platform and converted them into Python DataFrames. The private datasets are collected during the operation of the Vehicle Register (VReg) and Identity Card Application Management System (ICard) systems. Given that both systems are integral components of the national data assets, releasing this information to the public is not feasible. The VReg system provides IT services for vehicle-related customer service, while ICard provides services for

procedures/proceedings associated with personal identification documents such as Identity Card. The systems are based on a microservice architecture running on Kubernetes [1] containerized environment.

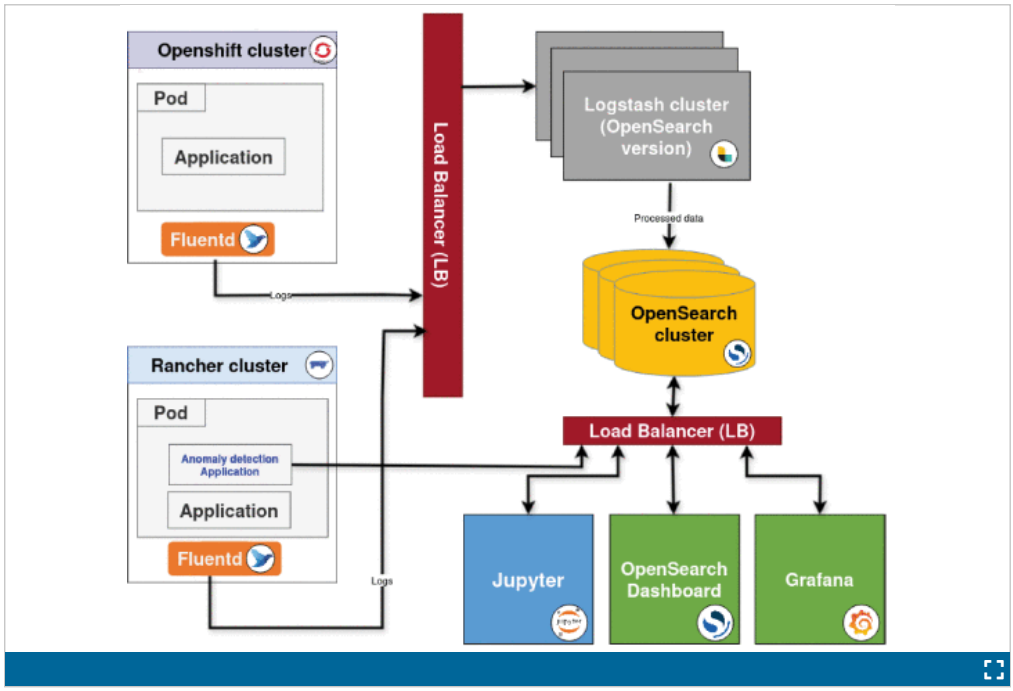


FIGURE 3.
Logs are gathered using Fluentd, transformed via Logstash, and ingested into OpenSearch. JupyterLab is used for advanced data analysis such as anomaly detection, whereas Grafana and OpenSearch Dasboards for data visualization and basic data analysis.

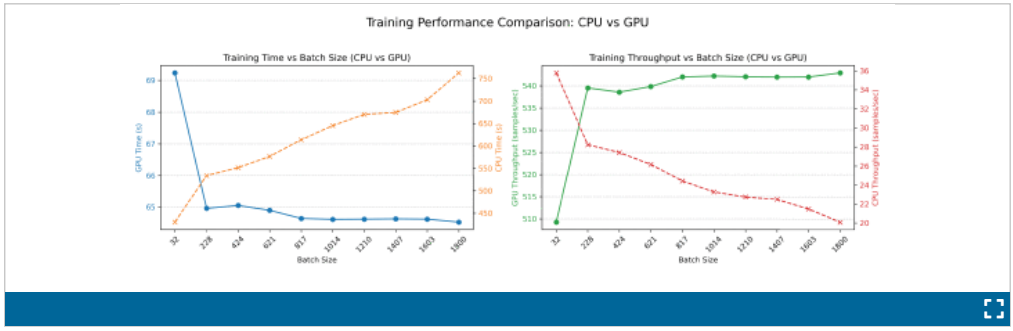


FIGURE 4.
The left subplot displays the total training time for both CPU and GPU across varying batch sizes, with separate y-axes used to illustrate the differing scales of each hardware configuration. The right subplot depicts training throughput, measured in samples per second, for both CPU and GPU, also utilizing separate y-axes to reflect their distinct performance ranges.

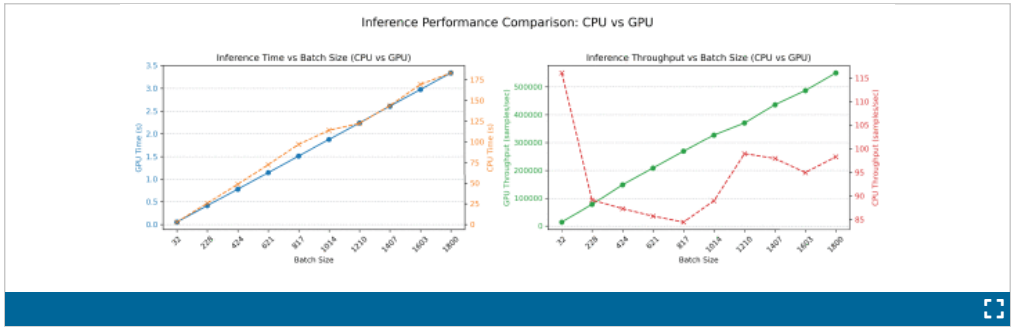


FIGURE 5.
The left subplot illustrates the total inference time, employing separate y-axes for CPU and GPU to highlight differences in scale. The right subplot displays throughput, expressed in samples per second, as a function of batch size.

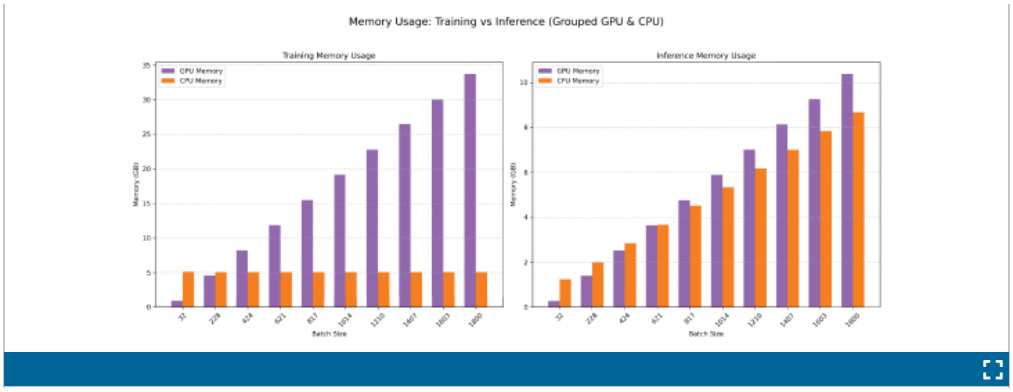


FIGURE 6.
The left subplot shows GPU and CPU memory consumption during training, while the right subplot displays memory usage during inference. Each bar represents the memory required for a given batch size, with GPU and CPU results grouped for direct comparison.

The first public BGL [23] dataset gathered over a 7-month period from the BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL) has been widely used in various research for log analysis, such as anomaly detection and log parsing, etc. The dataset contains over 4.7 million normal logs and more than 300 000 anomalies, which are already labeled.

The second publicly available Thunderbird dataset [23] was collected from the Thunderbird supercomputer system at Sandia National Laboratories (SNL) in Albuquerque, New Mexico. This dataset comprises over 200 million log entries, including more than 1 million events annotated as anomalies.

Datasets: To optimize the computational cost and time, we did not use the entire dataset. The training and testing datasets are as follows:

- **BGL**: Training data from before August 1, 2005, which includes 2 676 916 normal logs and 231 882 anomalies.
- **Thunderbird**: Training data from before November 15, 2005, which includes 2 843 286 normal logs and 66 198 anomalies.
- **VReg**: We retrieved the data from the OpenSearch log analysis platform [24] for the period between November 1, 2023, and December 12, 2023. The data were stored in the CSV format and subsequently labeled. Additionally, the first 12 days of the interval were duplicated and added to the dataset, extending the end date of the interval to January 5, 2024.
- **ICard**: We also retrieved data from OpenSearch for the period between February 18 and February 23, 2024. The data were stored in CSV format and labeled. Additionally, the first five days of the interval were duplicated and added to the dataset, extending the end date of the interval to March 4, 2024.

The reduced datasets used for training and evolution are presented in Table 3. These datasets where also used to train the LogTokenizer.

Experiment setup: In our experiments, we utilized a Kubernetes containerized environment with the following specifications: 32 vCPU, 120 GB RAM, and an NVIDIA A100 40 GB GPU. The F1 score, Accuracy, Recall, and Precision metrics were used to evaluate the performance of the experiments. We distinguished between true positives (TP) for anomalous logs, true negatives (TN) for normal logs, and false positives (FP), which represent instances classified as anomalous, but in fact, they are normal, and false negatives (FN) for the rest. The formulas used to evaluate the experiment are as follows:

$$\begin{aligned} Recall &= \frac{TP}{TP+FN} \\ Precision &= \frac{TP}{TP+FP} \\ F1 &= 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall} \\ Accuracy &= \frac{TP+TN}{P+N} \end{aligned}$$

View Source

Given that existing methods classify entire log sequences as anomalous, whereas we identify each log message as problematic within a log sequence, and considering that the two private datasets are in JSON semi-

structured format, the comparison is not directly equivalent. To enable a more suitable comparison, we converted the two private datasets into unstructured text format. For comparison, we used publicly accessible source code from [25]. We conducted the experiments using the datasets introduced in Section IV and applied the workflow shown in Figure 1. The results of the baseline methods are summarized in Table 4. Regarding recall, the baseline methods performed exceptionally well on the Thunderbird and ICard datasets. However, their performance on VReg and BGL datasets lagged behind the other two. Overall, LogAnomaly and DeepLog outperformed LogBert across all the datasets.

TABLE 4 Performance Evaluation of Baseline Methods Across Different Datasets

Dataset	Method	F1	Precision	Recall	Accuracy
BGL	LogBert	0.86	0.83	0.89	0.94
	LogAnomaly	0.86	0.85	0.87	0.94
	DeepLog	0.86	0.87	0.82	0.94
Thunderbird	LogBert	0.96	0.99	0.92	0.96
	LogAnomaly	0.99	0.93	0.99	0.95
	DeepLog	0.95	0.91	0.99	0.94
VReg	LogBert	0.68	0.68	0.67	0.65
	LogAnomaly	0.91	0.85	0.98	0.90
	DeepLog	0.93	0.88	0.98	0.92
ICard	LogBert	0.94	0.98	0.89	0.94
	LogAnomaly	0.99	0.98	0.99	0.99
	DeepLog	0.98	0.98	0.99	0.98

A. Evaluation of Our Methods

To conduct our experiments, we used the same datasets as the baseline methods described in Section IV for both the training and evaluation.

1) LMLAD Results

In the Anomaly detection workflow, the first step of LMLAD is the log message representation phase. As mentioned previously, we used the LogTokenizer with a maximum sentence length of 512 and the Byte-Pair Encoding (BPE) tokenizer type. For model training (Modeling phase), we utilized the Trainer module from the Hugging Face Transformers open-source library and Weights & Biases (WandB) to track and visualize the experiments. In the anomaly detection phase, as mentioned previously, the threshold was determined using the distribution plot of the BLEU score. In our experiment, we used kernel density estimation to estimate the probability density function (PDF) of the BLEU scores. From this distribution, we identified the 100 lowest distinct minima and the 100 highest distinct maxima as threshold candidates to construct a list of potential anomaly thresholds, called the BLEU score threshold list. These thresholds were then used to evaluate classification performance metrics, such as the F1 score and accuracy, between different BLEU scores. The optimal threshold corresponds to the index in the BLEU score threshold list where both the F1 score and the accuracy reach their maximum values, as illustrated in Figure 7.

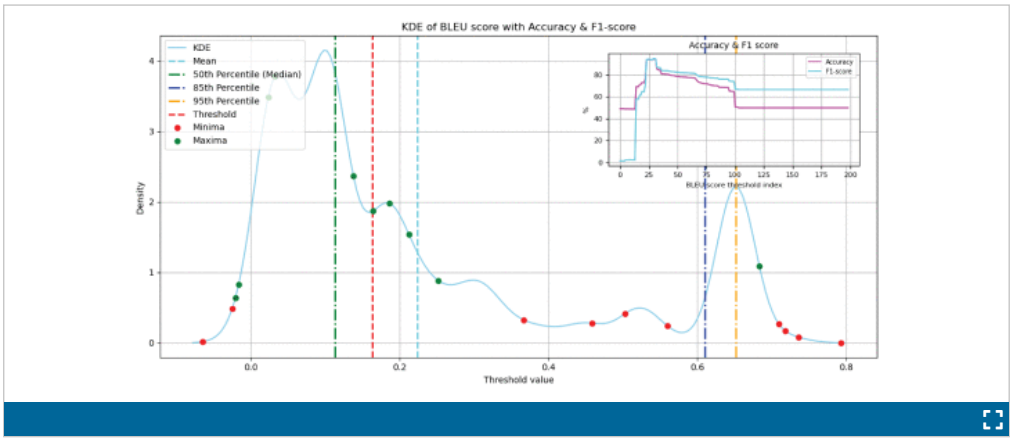


FIGURE 7.
Kernel Density Estimate (KDE) of the BLEU score distribution, annotated with percentiles and the calculated threshold. The inset displays the variation in accuracy and F1 score as functions of the BLEU threshold in the Thunderbird experiment using LMLAD.

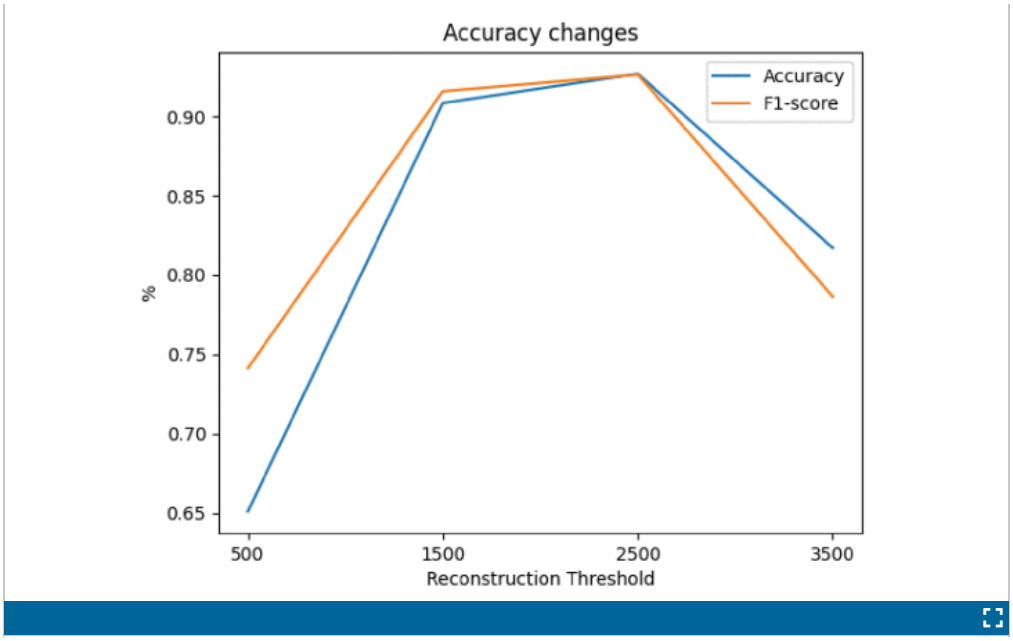


FIGURE 8.
Change in accuracy and F1 score in terms of the reconstruction threshold of the BGL experiment with CAELAD.

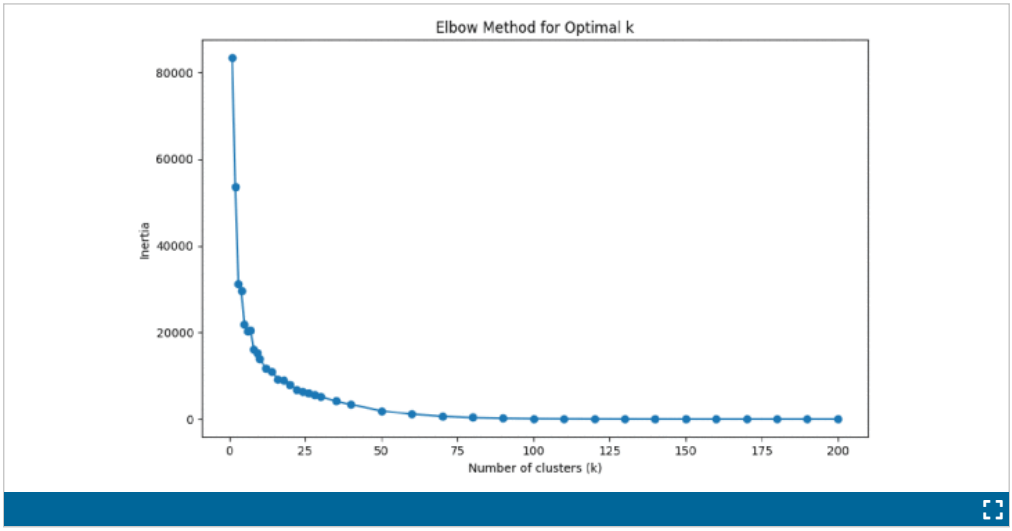


FIGURE 9.
Elbow method for determining the optimal number of clusters in the BGL experiment with SCLAD.

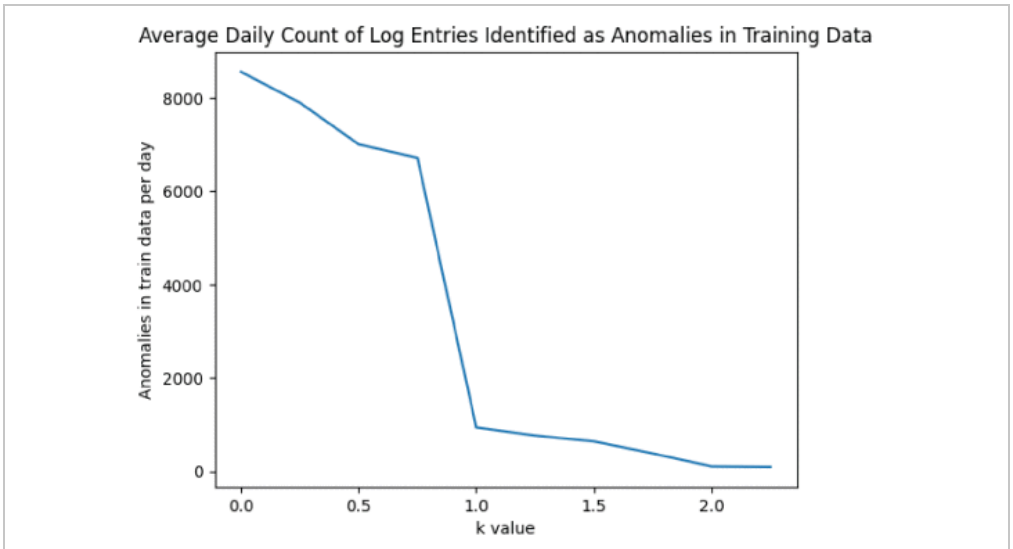


FIGURE 10.
Average daily count of log entries identified as anomalies in training data BGL in the function of k value.

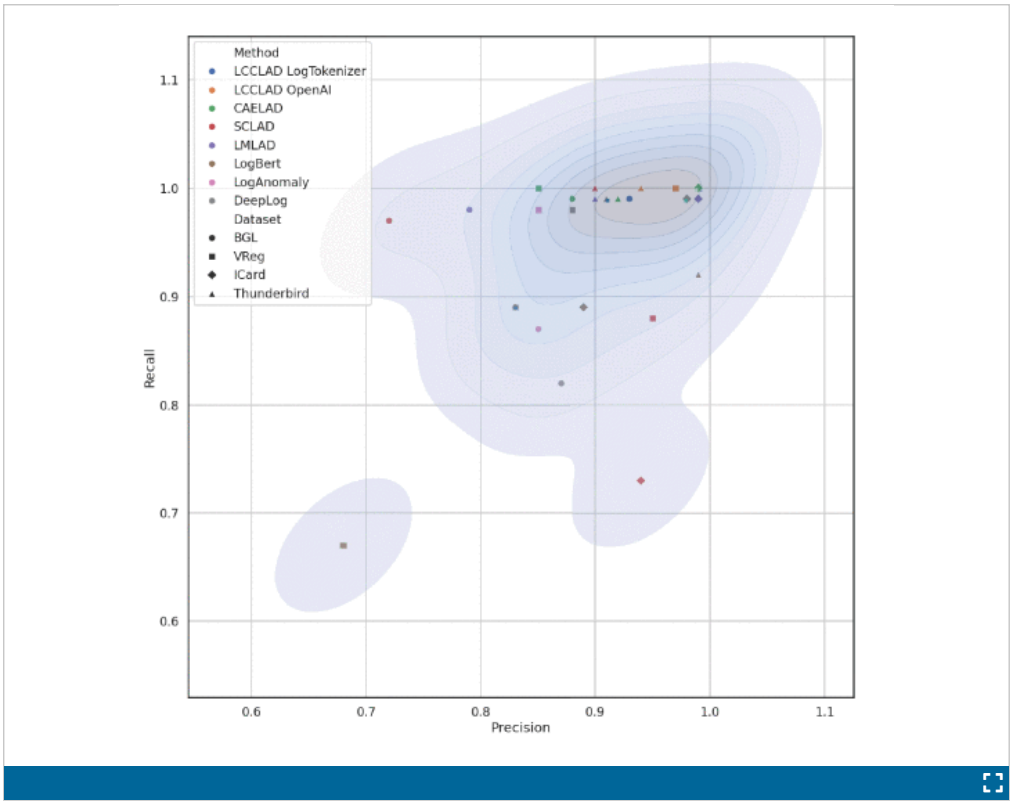


FIGURE 11.
This plot illustrates the relationship between precision (x-axis) and recall (y-axis) across all experiments conducted on the BGL, Thunderbird, VReg, and ICard datasets. The contour lines from the KDE classify the algorithms in terms of performances.

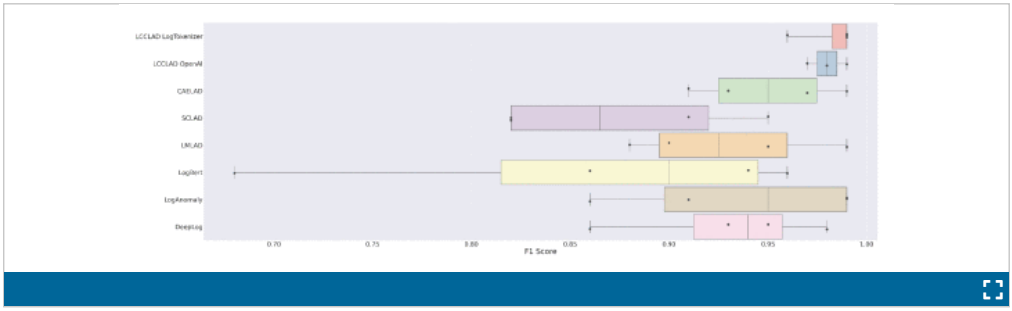


FIGURE 12.
The F1 scores for all experiments conducted on the BGL, Thunderbird, VReg, and ICard datasets.

Based on recall, LMLAD performed better on the BGL, Thunderbird and ICard datasets, demonstrating that LMLAD is effective for both unstructured and semi-structured data as shown in Figure 13. The results are summarized in Table 5.

TABLE 5 Comparison of Different Anomaly Detection Methods Across Datasets

Dataset	Method	F1 Score	Precision	Recall	Accuracy
BGL	LMLAD	0.88	0.79	0.98	0.86
	LCCLAD (OpenAI)	0.99	0.99	1.00	0.99
	LCCLAD (LogTokenizer)	0.97	0.94	0.99	0.97
	CAELAD	0.93	0.88	0.99	0.93
	SCLAD	0.83	0.72	0.98	0.80
Thunderbird	LMLAD	0.95	0.90	0.99	0.94
	LCCLAD (OpenAI)	0.97	0.94	1	0.99
	LCCLAD (LogTokenizer)	0.99	0.98	0.99	0.99
	CAELAD	0.97	0.92	0.99	0.96
	SCLAD	0.95	0.90	1.00	0.94
VReg	LMLAD	0.90	0.83	0.89	0.89
	LCCLAD (OpenAI)	0.99	0.97	1.00	0.99
	LCCLAD (LogTokenizer)	0.99	0.99	1.00	0.99
	CAELAD	0.91	0.85	1.00	0.92
	SCLAD	0.92	0.95	0.89	0.92
ICard	LMLAD	0.99	0.99	0.99	0.99
	LCCLAD (LogTokenizer)	0.99	0.99	1.00	0.99
	CAELAD	0.99	0.99	1.00	0.99
	SCLAD	0.83	0.95	0.73	0.84

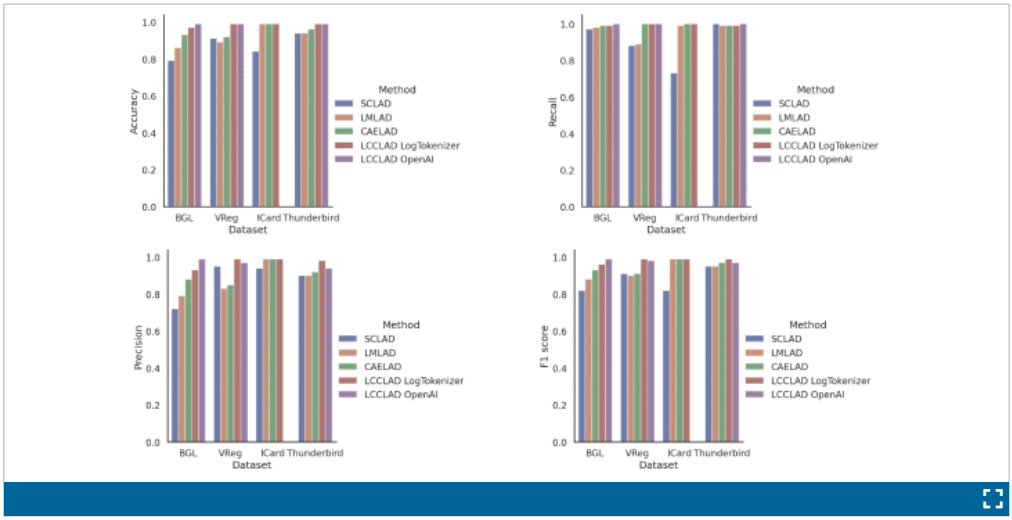


FIGURE 13.
Performance metrics of LMLAD, LCCLAD, CAELAD, and SCLAD experiments conducted on BGL, Thunderbird, ICard, and VReg datasets.

When operating IT systems, it is crucial to minimize the occurrence of false alarms. Although LMLAD generally generates a few false alerts during normal operations, false positives and negatives may occasionally be sent out when encountering anomalous data. This implies that system operators may overlook significant system errors, thereby leading to potential issues and risks. LMLAD does not require data labeling or log transformation (data pre-processing), but it demands substantial computing resources and a considerable amount of time for training and inference. Consequently, this method cannot be used without the use of a GPU even for prediction tasks, because running these tasks on a CPU would take longer, which is unacceptable in a production environment handling more than 30 000 logs per second. However, if the data are stored in the JSON format and can be searched efficiently, then the number of logs can be significantly reduced to errors and warning messages. Using these reduced data, we can significantly accelerate the anomaly detection process.

To assess the computational requirements of the LMLAD model under high workload conditions, we conducted two experiments using the environment defined in section IV: one focusing on training and the other for inference. During training phase, we measured the model's performance using both GPU and CPU across a range of batch sizes, as shown in Figure 4. For the GPU, the training step time remained nearly constant beyond a batch size of 817, and throughput showed no significant changes, indicating efficient scaling. In contrast, the CPU exhibited increasing training step times and decreasing throughput as the batch size grew. This suggests that the CPU struggles to handle the increased computational load required for forward passes, loss computation, backpropagation, and parameter updates. In general, the GPU significantly outperformed the CPU, achieving execution times approximately an order of magnitude faster, as shown in Figure 4. For the inference phase, we utilized the same set of batch sizes as in the training. We measured both inference latency and throughput using both a CPU and a GPU, as shown in Figure 5. The results illustrates a substantial performance gap between the CPU and GPU, with inference latency differing by approximately

two orders of magnitude and throughput by about five orders of magnitude similar to the differences observed during the training phase. In terms of memory usage, GPU memory consumption during training increases approximately linearly with batch size, while RAM usage remains relatively constant as shown in Figure 6. In contrast, both GPU memory and RAM usage during inference exhibit a linear growth with increasing batch size. However, the overall memory demands are significantly lower than those observed during training. As mentioned above, inference consistently requires less GPU memory and RAM compared to training, highlighting its lower resource footprint.

2) LCCLAD Results

The performance of LCCLAD was evaluated on the four benchmark datasets mentioned in Section IV: BGL, Thunderbird, VReg, and ICard. The outcomes are summarized in Table 6. Each dataset was processed using both methods (OpenAI and LogTokenizer), except for the private ICard dataset, which was processed only with LogTokenizer due to data protection concerns. As we mentioned previously the OpenAI's text-embedding-3-small model [28] produces fixed-length 256-dimensional vectors. Vector lengths were as follows for LogTokenizer: 336 for BGL, 266 for Thunderbird, 62 for VReg, and 50 for ICard (with zero padding applied to match the longest log vector in each dataset). The text-embedding-3-small method is denoted as OpenAI in Table 5.

TABLE 6 Training and Evaluation Datasets with Cluster and Anomaly Numbers

Dataset	#logs	method	#clusters	#training logs	#test logs	#anomalies in test
BGL	2,676,916	OpenAI	18	400,000	450,000	94,000
		LogTokenizer	17	425,000	170,000	67,000
Thunderbird	2,843,286	OpenAI	11	275,000	275,000	20,000
		LogTokenizer	21	210,000	210,000	15,000
VReg	914,537	OpenAI	13	325,000	325,000	13,000
		LogTokenizer	12	300,000	300,000	39,000
ICard	168,045	LogTokenizer	9	53,000	90,000	28,000

The vector selection process for both training and testing involved clustering the data and sampling a predefined number of vectors from each cluster. The number of clusters was determined using the elbow method and, when necessary, validated with the silhouette score. Table 6 lists the number of clusters for each database. Training vectors emphasized normal behavior, while test vectors intentionally included a higher proportion of anomalies. For example, if a cluster contains 15,000 anomalous and 70,000 normal log vectors, and the training and test set sizes are both set to 25,000, the training set is populated with 25,000 randomly selected normal vectors. From the remaining pool, 10,000 additional normal and 15,000 anomalous vectors are selected for the test set. This ensures balanced evaluation and allows the model's sensitivity to anomalies to be meaningfully assessed.

To evaluate consistency, the experiment was repeated ten times with different random seeds. Cosine similarity between test vectors and their ten nearest neighbors in the training set was used to compute detection scores. A high similarity threshold (typically $\tau > 0.9$) was set to distinguish normal from anomalous logs.

The LogTokenizer-based implementation consistently outperformed the OpenAI baseline in computational efficiency and showed comparable or superior accuracy. For instance, the Thunderbird dataset achieved an F1-score of 0.99 using LogTokenizer versus 0.97 with OpenAI. On the BGL dataset, LogTokenizer achieved an F1-score of 0.927. These results demonstrate that LogTokenizer offers a highly effective, resource-efficient alternative for real-time anomaly detection in log data. The results are summarized in Table 5.

3) CAELAD Results

In line with the processes used in LMLAD and LCCLAD, the anomaly detection workflow begins with the log message representation phase. In this step, log messages were transformed into fixed-length vector representations (length of 128) using the previously introduced LogTokenizer. The next phase, the modeling step, involved training the CAELED model for three epochs while calculating the mean squared error between the reconstructed and original data points. This allows the model to learn normal patterns within log data. In the final phase of the workflow, the reconstruction loss was computed by comparing the test data (test logs) with the predicted reconstructed logs. The threshold for the reconstruction loss is determined to optimize the recall for each experiment, as illustrated in Figure 8, and a summary of the experimental results is provided in Table 5. One of the advantages of CAELAD is its capability to run on both GPU and CPU, requiring approximately 1.5 GB of GPU memory. However, a GPU is necessary for production use.

4) SCLAD Results

Unlike other anomaly detection algorithms, the initial step in this workflow is log transformation, where log messages are parsed using the Drain algorithm, as described earlier. Drain extracts the constant parts of the log messages by removing variable components that are deemed non-informative for the analysis. This allows for the meaningful application of the SCLAD by focusing solely on the invariant parts of the log messages. In

the subsequent step, the Tf-Idf transformation is comparable to baseline methods in that it converts log messages into vector representations.

However, the modeling step diverges significantly from the baseline approaches. It leverages K-Means clustering to group vectorized log messages using the elbow method employed to determine the optimal number of clusters, as shown in Figure 9. In the anomaly detection phase of the workflow, the optimal k values are calculated to define the threshold for classifying a vectorized log entry as an anomaly, as demonstrated in Figure 10. An unsupervised training approach is utilized, where a database containing only normal (non-anomalous) logs is used for training. The k parameter selection follows the elbow method principle applied to the anomaly detection phase: as k increases, fewer log entries are classified as anomalies because more data points fall within the expanded cluster boundaries ($k\times$ standard deviation). The optimal k is selected at the point where this decreasing trend shows the sharpest change in slope, as illustrated in Figure 10.

This approach balances detection sensitivity with operational constraints, improving precision while accepting reduced recall in the context of SCLAD's pre-filtering role. The results of the experiments are summarized in Table 5. SCLAD demonstrates variable performance across datasets, with recall ranging from 0.73 to 1.00 and precision from 0.72 to 0.95. The method achieves excellent performance on Thunderbird (recall: 1.00, precision: 0.90) and BGL (recall: 0.98, precision: 0.72), while showing more moderate results on VReg (recall: 0.89, precision: 0.95) and ICard (recall: 0.73, precision: 0.95). As a pre-filtering method, the recall limitations on some datasets (particularly ICard) indicate that 27% of true anomalies would be missed in the initial screening stage. However, SCLAD's primary advantages remain its computational efficiency and real-time processing capabilities without GPU requirements. The dataset-dependent performance suggests that SCLAD is most suitable for log types similar to Thunderbird and BGL, where it can effectively serve as a first-stage filter while maintaining high anomaly detection rates.

SECTION V.
Comparative Analysis

Overall, all anomaly detection methods performed reasonably well on the investigated datasets; however, when considering recall alongside the complexity of the algorithms, we observe slight differences, which can be derived from the results summarized in Figure 13. SCLAD's performance lags behind the other algorithms, but it is notably fast and requires minimal computing resources (CPU, RAM) shown in 7.

In the Anomaly detection workflow, only SCLAD utilizes Drain for log parsing, whereas the other methods rely on raw data (JSON, free text) to create vector representations of the logs. LCCLAD was tested with both OpenAI vector embeddings and token vectors generated by LogTokenizer, CAELAD was tested using token vectors produced by LogTokenizer, and LMLAD also utilized LogTokenizer.

In addition, we measured the training and inference times, along with the GPU usage for the algorithms. The results are summarized in Table 7.

TABLE 7 Performance Comparison Across Different Datasets and Anomaly Detection Methods. We Compared the Training Time, Inference Time, and GPU Usage for Each Method. The Times in the Table Represent the Processing of 1 000 Logs in Seconds. GPU Usage is Measured During Both the Training and Inference Processes

Dataset	Method	Recall	Time [sec/1k log]		GPU Usage [GB]	
			Train	Inference	Train	Inference
BGL	LCCLAD LogTokenizer	0.99	0.23	0.92	n/a	n/a
	LCCLAD OpenAI	1.00	1.556	2.22	n/a	n/a
	CAELAD	0.99	0.02	0.014	1.40	1.80
	SCLAD	0.97	0.27	0.002	n/a	n/a
	LMLAD	0.98	4.19	6.67	29.90	7.40
	LogBert	0.89	0.74	2.47	5.90	5.90
	LogAnomaly	0.87	1.05	37.35	1.30	0.3
	DeepLog	0.82	1.58	34.37	0.20	0.20
Thunderbird	LCCLAD LogTokenizer	0.99	0.23	0.72	n/a	n/a
	LCCLAD OpenAI	1	1.459	2.17	n/a	n/a
	CAELAD	0.99	0.05	0.025	1.80	2.20
	SCLAD	1.00	0.23	0.05	n/a	n/a
	LMLAD	0.99	2.80	12	33	10.60
	LogBert	0.92	0.41	0.85	5.8	4.8
	LogAnomaly	0.99	1.07	17.67	1.5	0.90
	DeepLog	0.99	0.24	13.15	0.30	0.20
VReg	LCCLAD LogTokenizer	1.00	0.06	0.77	n/a	n/a
	LCCLAD OpenAI	1.00	1.66	2.01	n/a	n/a
	CAELAD	1.00	0.05	0.025	1.40	1.80
	SCLAD	0.88	0.04	0.005	n/a	n/a
	LMLAD	0.89	0.81	6.22	29.90	1.90
	LogBert	0.67	1.38	15.35	3.40	3.40
	LogAnomaly	0.98	1.24	85.10	0.2	0.30
	DeepLog	0.98	2.84	77.20	0.20	0.2
ICard	LCCLAD LogTokenizer	1.00	0.152	0.172	n/a	n/a
	CAELAD	1.00	0.03	0.026	1.40	1.80
	SCLAD	0.73	0.10	0.005	n/a	n/a
	LMLAD	0.99	167.99	1.15	29.90	2.30
	LogBert	0.89	1.38	0.75	1.40	4.00
	LogAnomaly	0.99	0.30	1.40	0.2	0.2
	DeepLog	0.99	0.27	1.25	0.20	0.20

Figure 11 illustrates the precision versus recall values for all experiments, providing a visual representation of the performance of the algorithms. The contour plots for the Kernel Density Estimation (KDE) of precision-recall pairs separate the algorithms in terms of performance, including LCCLAD LogTokenizer, LCCLAD OpenAI, CAELAD, SCLAD, LMLAD, LogBert, LogAnomaly, and DeepLog. The contour line shows the best kernel density estimations by classifying the different algorithms on the BGL, VReg, and ICard datasets. The algorithms positioned near the center of the main circle, such as LCCLAD OpenAI, LCCLAD LogTokenizer, LMLAD, CAELAD, LogAnomaly, and DeepLog, have recall and precision values close to one, indicating robust performance.

Figure 12 presents the F1 scores of the experiments conducted on the same BGL, Thunderbird, VReg, and ICard datasets. According to the performance illustrated in this figure, the methods can be classified into three distinct groups based on our evaluation metrics:

1) Top-performing algorithms: LCCLAD LogTokenizer, LCCLAD OpenAI, CAELAD

2) Mid-performing algorithms: LMLAD, LogAnomaly, DeepLog

3) Low-performing algorithms: SCLAD, LogBert

Alongside the evaluation metrics, another crucial measurement is the computing resource requirements. We repeated the experiments three times and recorded the maximum GPU usage and maximum time taken during both training and inference.

Figure 15 depicts the time taken during training and the inference, plotted against the achieved recall. In both cases (training represented by circles and inference represented by rectangles in the figures), the KDE contour plots differentiate the algorithms based on their time and recall. Figure 16 shows the algorithms whose training or inference time was less than 1 minute. As shown in the figure, only CAELAD and SCLAD achieved both training and inference times less than 1 minute. It is important to note that these times were measured using a single 40 GB A100 GPU. In practical applications, the performance can be scaled using multiple GPU instances, potentially reducing the runtime of more complex algorithms while handling the same volume of input data. Another method that nearly achieved training and inference times under 1 minute was LCCLAD using the LogTokenizer. In this case, only the BGL training exceeded 1 minute. As previously mentioned, most algorithms require a GPU. Figure 17 illustrates the GPU usage of the algorithms during training and inference, plotted against the corresponding recall. As shown in the figure, LMLAD requires substantial GPU resources for training of approximately 30 GB. The overall GPU usage of the algorithms is illustrated in Figure 18. Except for LMLAD and LogBert, the other algorithms used less than 2 GB of GPU during the training and inference. Based on the findings discussed above, we developed the flowchart presented in Figure 14, which aids in selecting a suitable algorithm depending on the anomaly detection approach, log format, available computational resources, and the desired recall. The flowchart differentiates between two

types of anomaly detection methods: (1) event-trace-based algorithms, which analyze sequences of multiple log messages, and (2) event-level algorithms, which focus on individual log lines, incorporating their contextual log level. Another key factor considered is the log format, while existing methods are designed primarily for unstructured logs, our proposed approach is capable of handling both structured and unstructured formats.

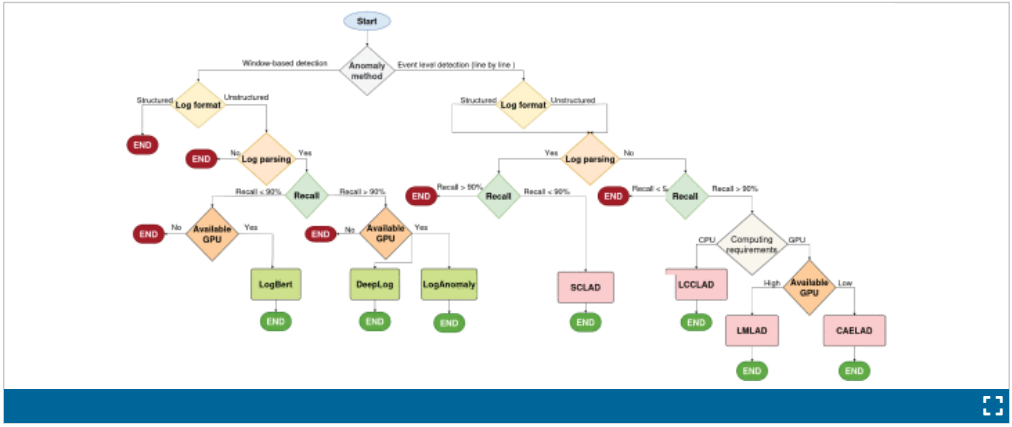


FIGURE 14.
Decision flowchart for selecting the appropriate log analysis method based on log format, parsing requirements, available GPU resources, and desired recall.

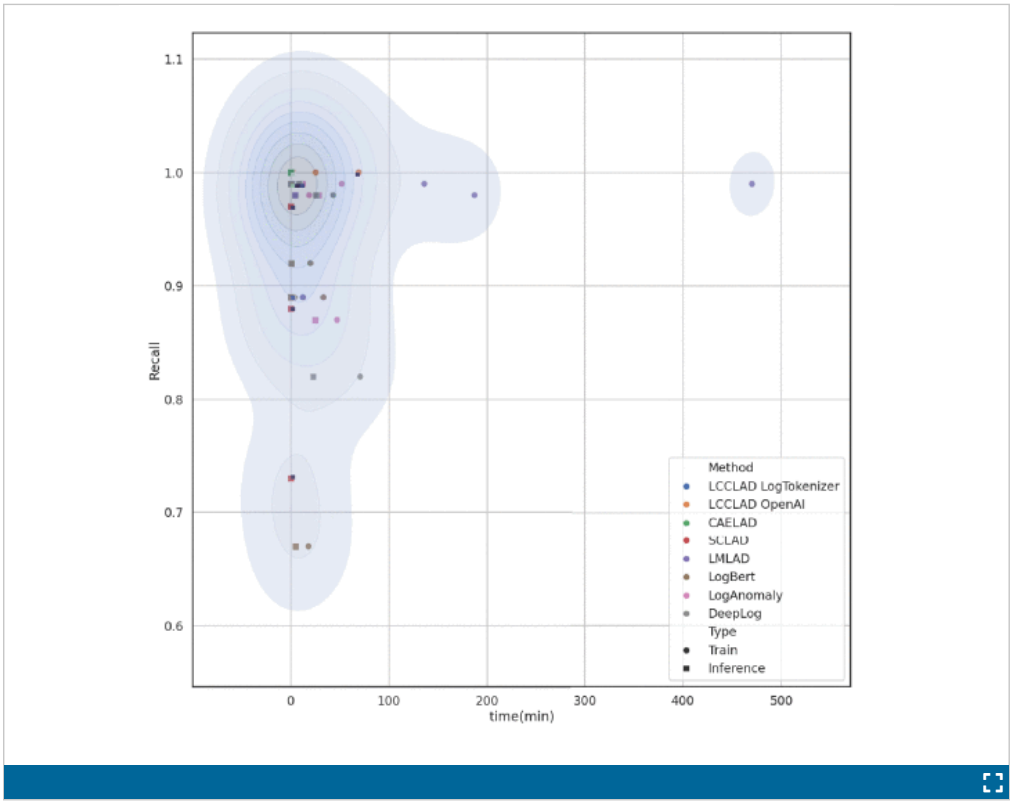


FIGURE 15.
This plot shows the relationship between training or inference time (x-axis) and recall (y-axis) for all experiments conducted on the BGL, Thunderbird, VReg, and ICard datasets. The contour lines from the KDE classify the algorithms based on training or inference time and performance, with circles representing training and squares representing inference.

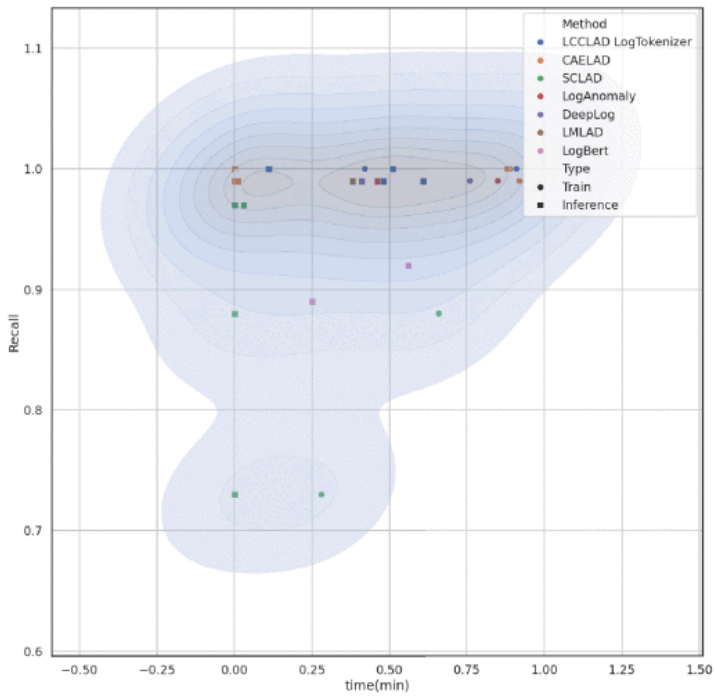


FIGURE 16.
This plot illustrates the relationship between training or inference time (x-axis) and recall (y-axis) for experiments with training and inference times of one minute or less, conducted on the BGL, Thunderbird, VReg, and ICard datasets. The contour lines from the KDE classify the algorithms based on training or inference time and performance, with circles representing training and squares representing inference.

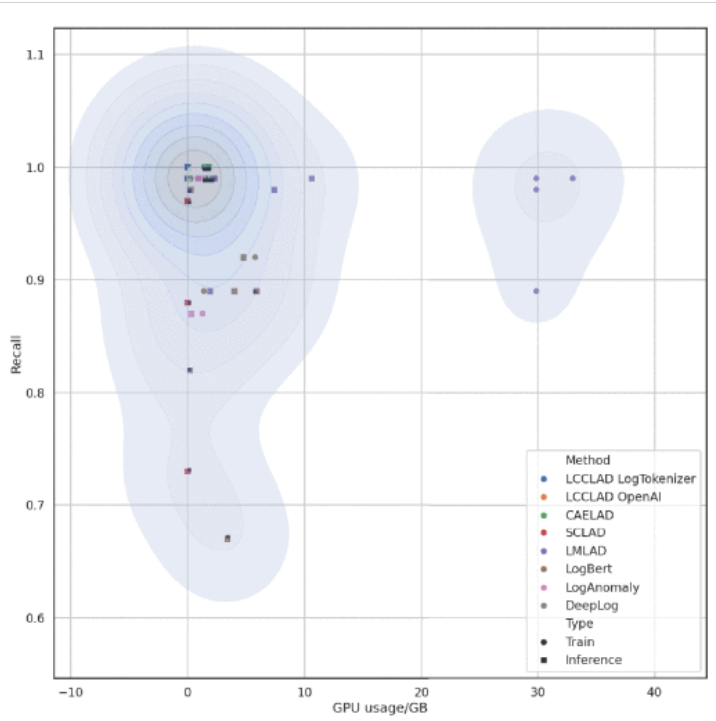


FIGURE 17.
This plot shows the relationship between GPU usage (x-axis) and recall (y-axis) for all experiments conducted on the BGL, Thunderbird, VReg, and ICard datasets. The contour lines from the KDE classify the algorithms based on GPU usage and performance, with circles representing training and squares representing inference.

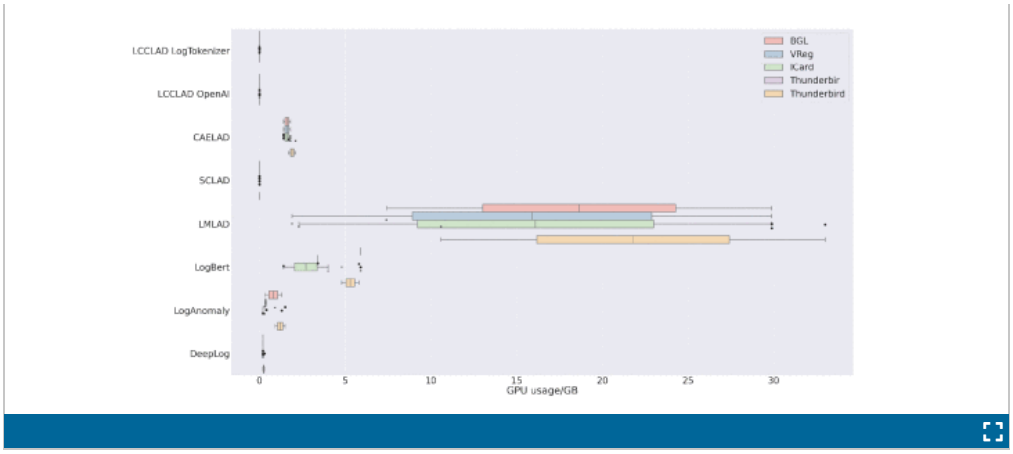


FIGURE 18.
This plot displays the GPU usage for all experiments conducted on the BGL, Thunderbird, VReg, and ICard datasets. Each point represents the GPU usage observed during either training or inference.

SECTION VI.
Cost Implications of Anomaly Detection Deployment

To evaluate the algorithm's suitability for production deployment, it is essential to consider the Total Cost of Ownership (TCO). In our context, TCO encompasses licensing fees, computing resource costs, operational expenses, personnel time, maintenance, and scalability. Given that current systems operate within a cloud-based infrastructure, comprising both public cloud services and on-premise Kubernetes clusters and considering the substantial volume of data to be processed, the algorithm must be capable of handling large-scale datasets efficiently. As noted previously, SCLAD does not require GPU acceleration, but it relies on an additional parsing step, which adds complexity to the workflow. In contrast, LMLAD, LCCLAD, and CAELAD algorithms streamline the log analysis process but incur additional costs, such as the need for GPUs and in the case of LCCLAD the need for vector storage systems in addition to the central log storage system. Both the training and inference phases of LMLAD are heavily dependent on the GPU infrastructure, which significantly increases operational and computational costs. When processing large volumes of data, multiple GPUs are required, further amplifying these expenses. Although CAELAD demonstrates more efficient GPU usage compared to LMLAD, it still incurs similar additional costs associated with GPU-based computation.

In contrast, LCCLAD does not require GPU resources but relies on vector embeddings or token vectors and a vector storage. GPU usage can be avoided when using vector embeddings via services such as OpenAI. However, this introduces additional costs, external dependency, and potential privacy concerns. The main advantages of OpenAI embeddings include usage-based pricing (cost per token) and eliminating local GPU infrastructure. However, these benefits come with trade-offs, such as recurring costs, data transfer over the network, and API rate limits, which can impact performance and data privacy. As an alternative, the LogTokenizer approach offers local, GPU-free operation with minimal performance degradation, approximately 1% difference in evaluation metrics compared to embedding-based methods. Additionally, if GPU resources are available, embedding services can be hosted locally using solutions such as Infinity embeddings [36], which can help balance cost, performance, and privacy considerations.

To improve the algorithm's performance, it is necessary to incorporate feedback from IT operators, which requires additional personnel time. While this adds complexity to the workflow, it can significantly reduce operational costs associated with service outages and potential Service Level Agreement (SLA) violations. Furthermore, integrating operator feedback enhances system proactivity and contributes to improved overall service quality.

SECTION VII.
Conclusion

To address the challenge of analyzing large volumes of free-text or semi-structured log data generated by various industry services, we investigated vector embeddings, token vectors, and log parsing for log message representation, and evaluated four different anomaly detection methods. We designed the anomaly detection workflow by considering industry trends in log analysis. Using the publicly available BGL and Thunderbird datasets, along with the private ICard and VReg datasets, we demonstrated the performance of the proposed LMLAD, CAELAD, LCCLAD and SCLAD algorithms. Subsequently, we compared these methods against three different baseline methods. The results indicate that the introduced algorithms CAELAD, LCCLAD, and LMLAD achieved an average F1 score 6.7% higher than baseline methods, with the exception of SCLAD, as demonstrated in Tables 4 and 5. Baseline methods show a roughly 3.4% higher average F1 score improvement over SCLAD, while in the case of CAELAD, LCCLAD, and LMLAD, this number is 10.4%. Additionally, we compared the vector embeddings from OpenAI with the token vectors from LogTokenizer and observed a 1% difference in the evaluation metrics. However, considering the cost and computational resources, the LogTokenizer is highly efficient, requiring fewer resources compared to OpenAI vector embeddings, and it also has the advantage of running locally.

In log analysis, scalability is crucial. We designed the SCLAD algorithm to require fewer computational resources than other algorithms, and it runs significantly faster. SCLAD shows dataset-dependent performance, achieving excellent recall on BGL (0.98) and Thunderbird (1.00) datasets, but more limited recall on VReg (0.89) and ICard (0.73). While its overall F1 scores lag behind other methods, SCLAD remains valuable as a pre-filtering method for specific log types, offering minimal complexity, absence of GPU requirements, and high processing speed.

The results indicate that the performance of the baseline methods can be exceeded without log parsing while using minimal GPU resources. If using a GPU is not an option, LCCLAD can still be utilized, but it necessitates the presence of a vector store database. Additionally, if issues arise with the operation of the vector store, SCLAD can still be utilized for suitable log types where it demonstrates high recall performance. In summary, we introduced four novel log anomaly detection algorithms that require no log parsing or labeled data, and demonstrated their effectiveness on real-world data. We found that simpler methods can sometimes match the accuracy of deep learning methods, providing practical options for different resource constraints.

Our next step is to further optimize CAELAD to enhance GPU utilization and processing speed. Additionally, we aim to evaluate the use of LMLAD vector embeddings alongside LogTokenizer and OpenAI embeddings. We also plan to explore hybrid methods, combining different approaches where data are initially processed by low-computation, simple techniques, such as CAELAD or SCLAD (for compatible log types). Based on theinitial results selected data can then be directed to more advanced and computationally intensive methods (such as LMLAD) for further analysis if necessary.

ACKNOWLEDGMENT

The authors thank Tamás Csiszár for contributing to the Python tool used to retrieve datasets from OpenSearch. The research reported in this publication, carried out by Pázmány Péter Catholic University and IdomSoft Zrt.

Appendix A
ICard and VReg Private Datasets

The ICard and VReg datasets referenced in this study are proprietary and stored in JSON format. Both datasets consist of log messages written in Hungarian (approximately 95%), with the remainder in English (approximately 5%).

Below are samples entries from the VReg and ICard datasets, showing only the message field for illustration. All identifying fields have been redacted for privacy.

VReg Example:

{

"_index": "REDACTED",

"_id": "REDACTED",

"_version": 1,

"_score": null,

"_source": {

"service.name": "REDACTED",

"docker": {

"container_id": "REDACTED"

},

"@version": "REDACTED",

"log.level": "REDACTED",

"message": "sikeres, válaszban

verzió/darabszám: 131/129",

"event.dataset": "REDACTED",

"@timestamp": "REDACTED",

"kubernetes": {

"container_image": "REDACTED",

"pod_name": "REDACTED",

"container_name": "REDACTED",

"namespace_name": "REDACTED",

"container_image_id": "REDACTED",

"host": "REDACTED",

"pod_id": "REDACTED",

"labels": {

"access/figy_mq": "REDACTED",

"app": "REDACTED",

"access/opentelemetry": "REDACTED",

"pod-template-hash": "REDACTED",

"access/figy_panic_rabbitmq":

"REDACTED",

"deploymentconfig": "REDACTED"

}

},

"process.thread.name": "REDACTED",

"tags": [

"REDACTED",

"REDACTED"

],

"time": "REDACTED",

"log.logger": "REDACTED",

"logtag": "REDACTED",

"stream": "REDACTED",

"ecs.version": "REDACTED"

},

"fields": {

"@timestamp": ["REDACTED"],

"time": ["REDACTED"]

},

"highlight": {

"log.level": ["REDACTED"]

},

"sort": ["REDACTED"]

}

ICard Example:

{

"_index": "REDACTED",

"_id": "REDACTED",

"_version": 1,

"_score": null,

"_source": {

"kind": "REDACTED",

"ecs.version": "REDACTED",

"tags": [

"REDACTED_TAG",

"REDACTED_PLATFORM"

],

"method": "REDACTED",

"error.type": "REDACTED",

"event.dataset": "REDACTED",

"span_id": "REDACTED",

"@timestamp": "REDACTED",

"kubernetes": {

"namespace_name": "REDACTED",

"container_image": "REDACTED",

"container_image_id": "REDACTED",

"pod_id": "REDACTED",

"host": "REDACTED",

"labels": {

"app": "REDACTED_APP",

"deployment": "REDACTED",

"open-telemetry": "REDACTED",

"pod-template-hash": "REDACTED",

"usingNamespace": "REDACTED",

"wp-mq": "REDACTED"

},

"container_name": "REDACTED",

"pod_name": "REDACTED"

},

"error.stack_trace": "app.error.

ErrorEvent:

ErrorEvent{details=[FAULT_CODE],

status=204 NO_CONTENT}\n

app.error.Processor.handleError(

ErrorProcessor.java:45)\n

app.logic.validate(ErrorHandler.

java:29)\n

app.logic.process(ErrorHandler.

java:38)\n

app.api.retrieve(DataHandler.

java:18)\n

\ldots ",

"process.thread.name": "REDACTED",

"service.name": "REDACTED",

"time": "REDACTED",

"logtag": "F",

"trace_flags": "REDACTED",

"message": "Error occurred in

application",

"class": "REDACTED"

},

"fields": {

"@timestamp": [

"REDACTED"

],

"time": [

"REDACTED"

]

},

"sort": [{

1750775924413

}]

}

The datasets were labeled according to the log.level field. Labeling of anonymous log entries was performed in collaboration with the development teams. Given that the proposed methods are based on unsupervised learning, these labels are utilized exclusively for evaluation purposes. Figures 19 and 20 represent time series representations of normal log message counts, denoted as doc_count, and anonymous log messages, denoted as anomaly.

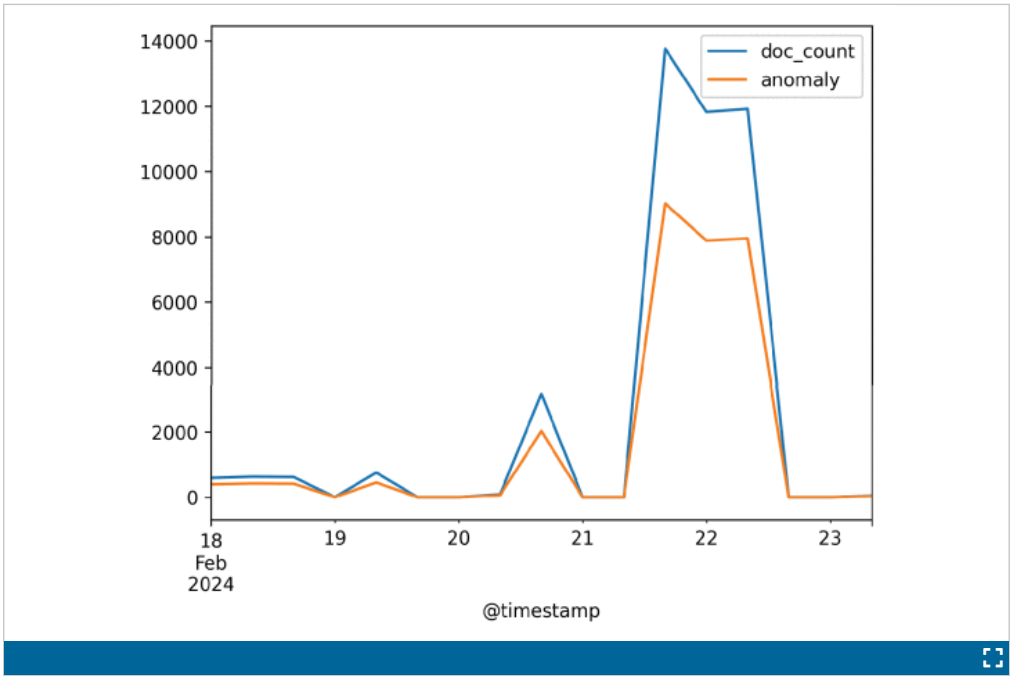


FIGURE 19.
Time series of log message counts for the ICard system. The plot presents the volume of normal log messages (doc_count) and anonymous entries labeled as anomalies (anomaly). A pronounced spike is observed on 21–22 February, indicating a period of elevated and potentially abnormal activity.

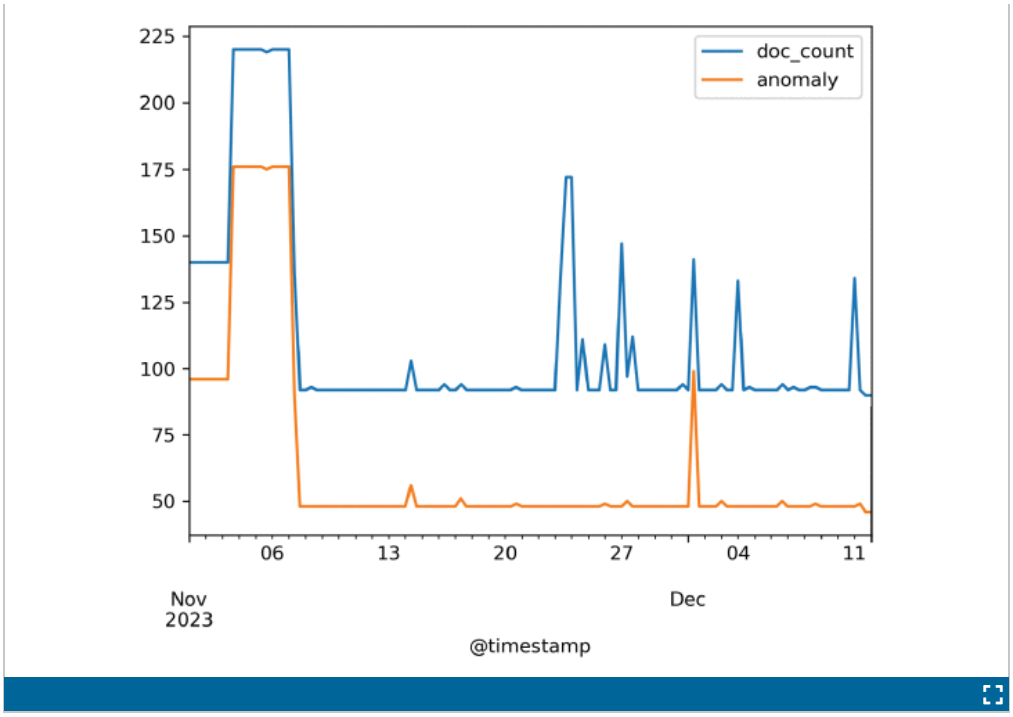


FIGURE 20.
Time series of log message counts for the VReg system. The plot presents the volume of normal log messages (doc_count) and anonymous entries labeled as anomalies (anomaly). Several local peaks in the anomaly series suggest irregular deviations in system behavior.

Figures 22 and 21 illustrate the anonymous log entries for the ICard and VReg systems, respectively. As shown, the VReg system exhibits fewer anomalies, both in terms of total count and frequency. In contrast, the ICard system shows a significantly higher volume of anomalous entries.

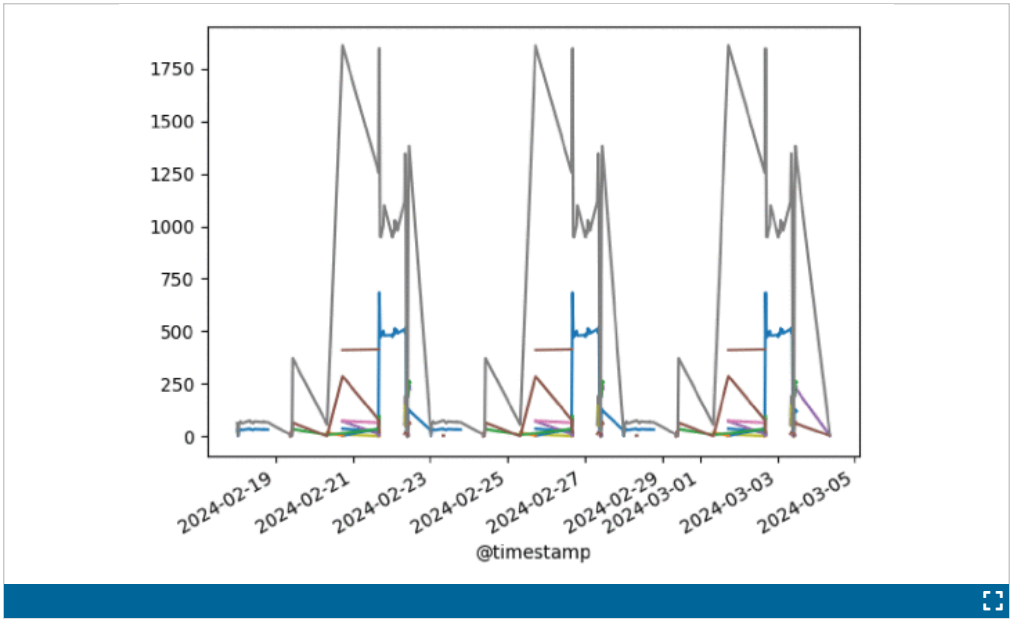


FIGURE 21.
Time series of anomalous log entries from the ICard system. Distinct types of anomalies are represented using different colors to highlight their distribution and variation.

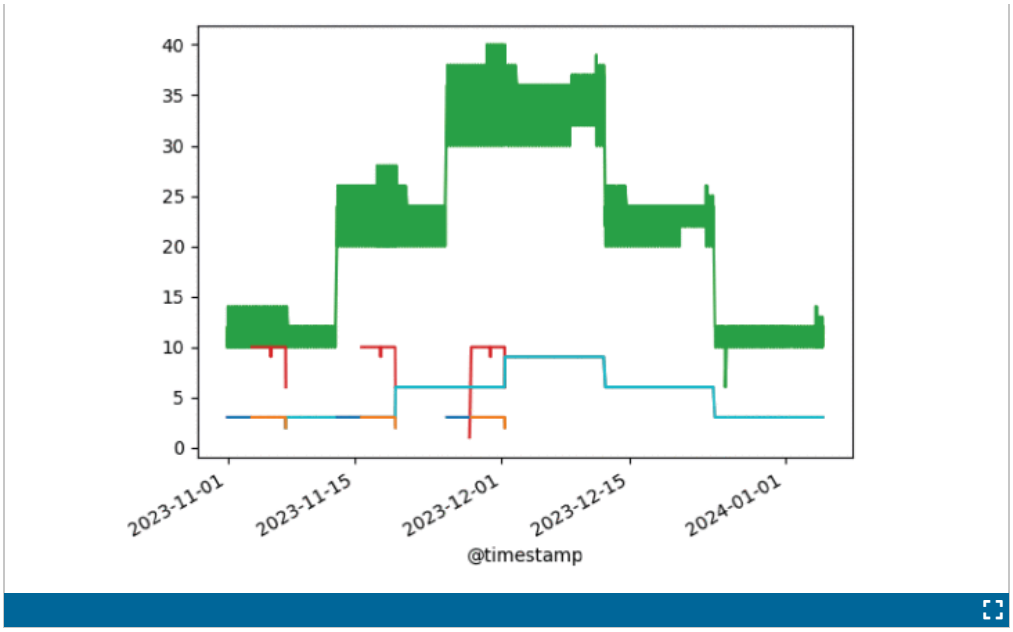


FIGURE 22.
Time series of the anomalies log entries of the VReg system. Distinct types of anomalies are represented using different colors to highlight their distribution and variation.

Appendix B
Comparision of LogTokenizer and LLama Tokenizers

To evaluate the effectiveness of our LogTokenizer compared to the original Llama Tokenizer, we performed a performance comparison using the LMLAD anomaly detection model on the Thunderbird dataset. The evaluation considered key metrics including F1 score, recall, precision, and accuracy. The KDE of the BLEU score, F1 score, and accuracy are illustrated in Figure 23 and summarized in Table 8. As shown in Table 8, the LogTokenizer significantly outperforms the Llama Tokenizer. Although both tokenizers achieve similarly high recall (0.99), LogTokenizer demonstrates substantially higher precision (0.90 vs. 0.65), F1 score (0.95 vs. 0.78), and accuracy (0.94 vs. 0.73). These results suggest that LogTokenizer offers more reliable anomaly detection by reducing false positives and improving overall classification performance.

TABLE 8 Performance Comparison of the LMLAD Model on the Thunderbird Dataset Using Two Different Tokenization Methods: LogTokenizer and Llama Tokenizer

Dataset	Method	F1 Score	Precision	Recall	Accuracy
Thunderbird	LMLAD LogTokenizer	0.95	0.90	0.99	0.94
	LMLAD Llama Tokenizer	0.78	0.65	0.99	0.73

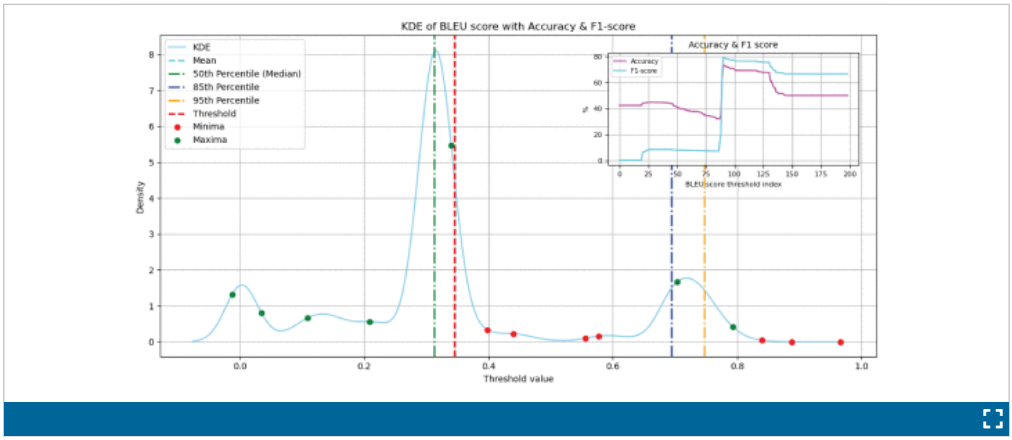


FIGURE 23.
Kernel Density Estimate (KDE) of the BLEU score distribution, annotated with percentiles and the calculated threshold. The inset displays the variation in accuracy and F1 score as functions of the BLEU threshold in the Thunderbird experiment using LMLAD and LlamaTokenizer.

Authors
Figures
References
Keywords
Metrics
Footnotes

Back to Results