UMEÅ UNIVERSITY

Anomaly Detection in Large-Scale Log Data

Parsing, Feature Extraction, and Unsupervised Learning.

Tomas Lestander

Tomas Lestander
Spring 2025
Degree Project in Computing Science and Engineering, 30 credits
Supervisor: Eunil Seo (Umeå University)
Extern Supervisor: Johan Eker (Ericsson)
Examiner: Henrik Björklund (Umeå University)
Master of Science Programme in Computing Science and Engineering, 300 credits

Abstract

This thesis addresses the challenge of processing and analyzing large-scale, unstructured log data, which is a task of growing importance in today's data centers. The thesis focuses on practical methods to extract meaningful features from large, unlabeled log datasets, with emphasis on using unsupervised learning techniques suitable for handling the scale and nature of the data effectively. Before applying the designed model to a real-world dataset, the accuracy of the designed model is tested on a publicly available dataset. The findings contribute to the field of log data analysis, offering insights into handling large datasets and highlighting potential areas for further research in anomaly detection and data processing techniques.

Acknowledgements

I would like to thank my external advisor *Johan Eker* and my internal advisor *Eunil Seo* for their guidance throughout the project. Their expertise and support have been invaluable. I also want to thank the students and teachers at Umeå University for all the support over the years, especially the student group *Boyzen* and the students *Abbe*, *Johan*, and *Elias*. And thanks to my family and friends for their encouragement, especially *arigatou* who inspired me on this path, and *Sarli*, *PJ* and *Andre* for all the mischief. And also my neighbors *Masih*, *Suleika* and *Samira* for all the fun at the student housing.

Contents

Glossary

NLP	Natural Language Processing
LLM	Large Language Model
ML	Machine Learning
DL	Deep Learning
NN	Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
GRU	Gated Recurrent Unit Neural Network
LSTM	Long Short-Term Memory Neural Network
OOV	Out-Of-Vocabulary
MLM	Masked Language Modeling
VHM	Volume Hypersphere Minimization
BERT	Bidirectional Encoder Representations from Transformers
SBERT	Sentence-BERT
AE	Autoencoder

1 Introduction

Data Centers produce large amounts of unlabeled log files every day, and this presents many challenges: the logs and their relationships are complex, unlabeled data does not have ground truths to compare against, the logs take up large amounts of disk space, and automated analysis becomes extremely expensive and time-consuming. Effective unsupervised methods are needed to capture intricate relationships between log messages, and the methods should also be efficient.

The logs contain valuable runtime information, recording events and actions that occur over time, and through the logs it is possible to trace events happening in the system, when they happen, and the locations of the events. Log analysis is an important step to improve the Quality of Service to the customers, by identifying and resolving issues, such as system failures, performance issues, or security threats, and can provide information on user interactions to optimize user experiences. Processing high-volume log data is time-consuming and efficiency is an important feature, possibly involving both simpler methods to indicate areas of abnormal behavior and advanced methods to detect complex information from the data.

Processing logs is typically performed in multiple steps before employing log anomaly detection methods, and the process usually begins with collecting and cleaning logs, parsing the logs into structured formats, partitioning large volumes into smaller manageable formats, and extracting numerical features to make the data usable in further analysis. There are many tools and techniques in log anomaly detection, ranging from simple statistical methods to advanced ML-based methods, and these methods can be further categorized as reconstruction-based or prediction-based.

1.1 Research Questions

The thesis aims to seek answers to the following research questions :

RQ1: *Feature Extraction in Unstructured Logs*: How can we extract meaningful features from unstructured log data to effectively use it for anomaly detection?

RQ2: *Handling Unlabeled Data*: With the absence of labeled data in log files, how can we adapt or develop unsupervised or self-supervised learning methods for accurate anomaly detection?

1.2 Challenges

Unstructured and Unlabeled Data: Addressing the challenges of unstructured log files and the lack of labeled data to train effective anomaly detection models.

Data Volume and Complexity: Managing the large volume and complexity of log data can affect the efficiency of processing and analysis.

Effective Feature Representation: Identifying and extracting features from log data that accurately represent the information for anomaly detection.

Model Selection and Efficiency: Finding the optimal model architecture and settings that balance detection accuracy and computational efficiency well.

1.3 Aims

This study aims to explore methods that can be applied to log data to better understand the data and hopefully to uncover interesting patterns. Since the data is unlabeled, the focus is on identifying suitable unsupervised or self-supervised methods.

1.4 Objectives

The study extends to further understanding anomaly detection methods and how they can be adapted to log data, which has unique, domain-specific characteristics.

This study is particularly interested in integrating both efficient methods alongside more demanding methods, since large-scale data is not suitable for computationally expensive methods.

To assess the effectiveness of the method, an evaluation benchmark is established to test the models' ability to detect log anomalies.

1.5 Contributions

This section describes the main research contributions of this study.

Methodological Framework: Developed a comprehensive pipeline for log data processing and anomaly detection using advanced methods such as transformer-based LLMs and RNNs.

Focus on Transparency: This paper aims to be as transparent as possible, with detailed descriptions of how experiments are set up, well-explained model architecture, and a clarification of the design choices and the reasoning behind them. The work does not aim to be a competitor among log anomaly detection methods, despite the high-achieving results on the publicly available dataset.

Insights for Future Research: Provided valuable insights into the potential and limitations of the underlying models used for log anomaly detection, setting the stage for future investigations.

1.6 Structure of Thesis

Section 2 provides background information on log processing, log embeddings, and techniques for detecting patterns and anomalies in system logs. Section 3 reviews related work, highlighting approaches and findings from previous studies. Section 4 presents the design of the proposed solution, explaining its components and structure. Section 5 describes the implementation details including tools and frameworks and how the design ideas were put together into working code. Section 6 presents information on the experiments, how they are conducted, and their respective results, while Section 7 is an evaluation of those results along with key findings and a discussion of performance and limitations. The last Section 8 concludes the report along with possible directions for future work and a personal reflection of the study.

2 Background

This section provides theoretical background on log anomaly detection, including common tools and techniques. It also provides information about OpenStack, which is one of the analyzed datasets used in this study.

2.1 OpenStack

OpenStack, launched in 2010 through a collaboration between Rackspace Hosting and NASA, is an open-source cloud computing platform, composed of services that work together to manage computing, storage, and networking resources, and is used to build scalable and flexible cloud environments, suitable for both private and public clouds. In OpenStack, a *service* refers to a software component that performs a specific function within the system, designed to operate independently but also with the ability to interact with other services to provide comprehensive cloud capabilities. The OpenStack cloud operating system is modular, such that users can deploy only the services they need, making it a flexible and efficient architecture [1][2][3]. Over the years, OpenStack has continuously grown in size and features, and the release version for the logs in this study has more than 40 available services [4].

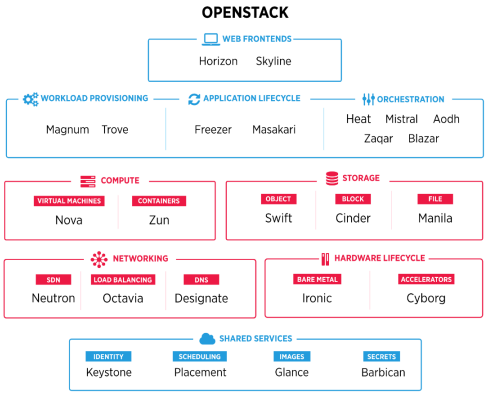


Figure 2.1: Simplified overview of OpenStack services [5]

Table 2.1 describes the key services and their responsibilities.

The OpenStack services have unique responsibilities, and they may also have sub-services and databases working together within each service. As such, the system is quite complex and the interactions may be hard to track with possibly simultaneous executions, causing interleaved messages both between services and within the services. Therefore, fixed ordering of generated OpenStack log messages is not always guaranteed [6]. This is important to consider when analyzing OpenStack log messages, because in the cases where there is no identifier for execution paths, the ordering of messages cannot be guaranteed and the messages may be interleaved and the analysis may require to view log messages in time windows. Interleaved messages are discussed in Cotroneo et al. [7], where the authors write that without a propagating

Service	Description
Horizon	User dashboard interacting with Keystone during log in.
Keystone	Authenticates the user and returns an authentication token for the user. The token is used in all subsequent requests.
Nova	Managing the life-time of VMs, including scheduling, creation and deletion of VMs.
Glance	Stores initial VM images and serves them to Nova.
Cinder	Persistent block storage to running VMs.
Neutron	Manages networking within and between services.
Swift	Distributed storage service, similar to cloud containers, that can scale horizontally with replicas for fault-tolerance.

Table 2.1: Descriptions of key OpenStack services.

ID, it is impossible to determine the correct order of events. To address this, they used time windows under the assumption that all relevant log messages for an event will appear within a specific time range. It is worth mentioning that some OpenStack log messages include a propagating ID during parts of the execution, making sequential analysis possible for those logs. The challenge is to understand how requests are propagated through the system, making the problem domain-specific.

Figure 2.2 illustrates the complexity of OpenStack and the interactions between its services when more services and subservices are incorporated.

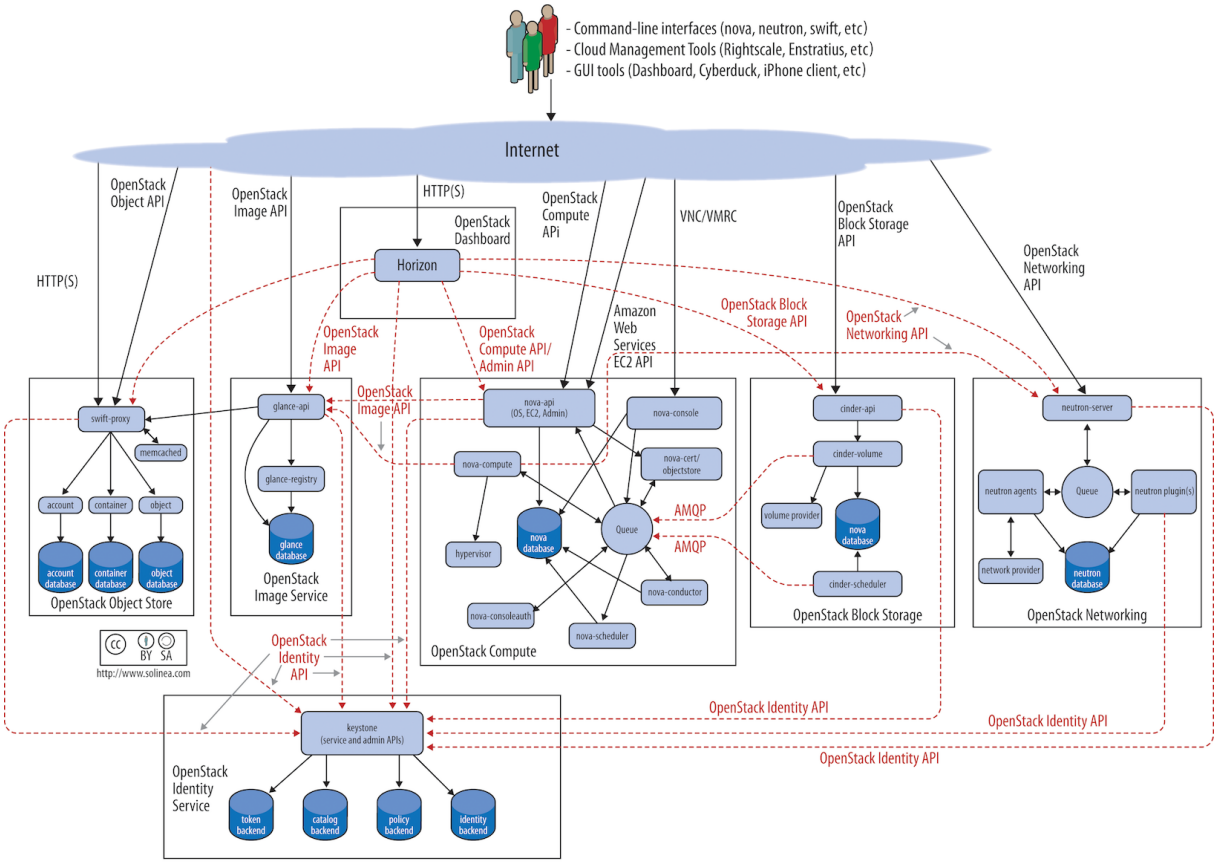


Figure 2.2: Openstack design[8] with common services and their interactions.

2.2 Log Messages

A log message is an informative text generated by a computer system to report system events, which can range from informative events such as users logging in or out, to more critical events such as intrusion detections, device failures, or database corruptions. Along with the actual message, a log often has additional information, such as timestamp, component that generated the message, and severity level. The severity level is based on how critical the event is and there are multiple standards with different sets of severity levels. Openstack uses the severity levels shown in Table 2.2, ranked from most severe (emergency) to least severe (info). Understanding the differences between log levels can be difficult, but in general, lower values indicate more severe conditions.

Value	Log level	Description
0	Emergency	The component or the entire system is unusable.
1	Alert	Immediate action is required, for example in the case of a corrupted database.
2	Critical	Critical conditions such as device errors.
3	Error	Error conditions that require attention.
4	Warning	Unexpected event that may require attention.
5	Notice	Normal but slightly more significant than *Info*.
6	Info	Informational messages.

Table 2.2: Descriptions of log severity levels.

2.3 Log Collection

The first step in the log processing pipeline is to collect the logs from sources such as servers, applications, networks, and security systems. As logs may arrive from different sources, the work may include aggregating them into a single, centralized system, making it more manageable to process the logs collectively. Notable challenges of collecting logs in large-scale systems are the velocities and volumes of the logs; the logs can quickly stack up in size, and it may be necessary to partition the log files into smaller time frames, such as hours or minutes. Another challenge is to determine the value of the logs by identifying redundant or otherwise unnecessary information in the logs.

2.4 Log Parsing

By log parsing, raw logs are classified and converted into a structured format, identifying different types of logs and providing them with identifiers. For each uniquely identified log message, variable information such as user IDs, IP addresses, etc. is replaced with placeholders, and the remaining parts of the message are referred to as the *template*. The parsing process can add additional information, such as timestamp, logging level, node component, and the extracted variable information. Figure 2.3 shows how a log message is classified with a log ID, a template, and variables.

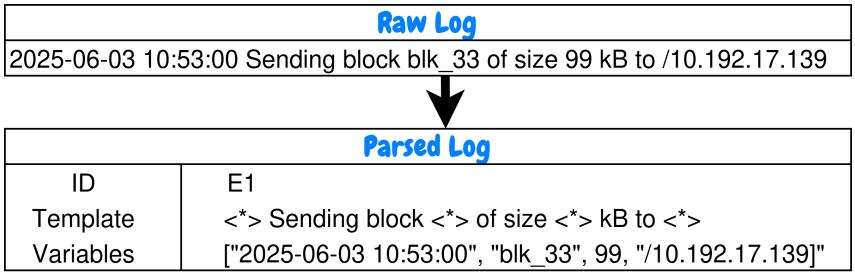


Figure 2.3: Extracted information after parsing a log message, classified into a group with a log ID and a template along with variable information.

When parsing general log messages, they rely on regular expressions and ad-hoc scripts provided by developers. These scripts separate log messages into different groups, where log messages in the same group have the same event template. In modern software engineering, even with these existing industrial solutions, log parsing is still a challenging task due to three main reasons: (1) the large volume of logs and thus the great effort on manual regular expressions; (2) the complexity of software and thus the diversity of event templates; and (3) the frequency of software updates and thus the frequent update of logging statements.

For large-scale systems, log messages may be written by many developers, and the message formats may vary between groups of developers. Zhu et al. [9] describes that, due to the complexity of log formats, log parsing used to rely heavily on writing regular expressions, and He et al [10] state that relying on regular expressions is an impractical approach, due to the large volume of logs, the diversity of event templates, the frequency of software updates, and therefore the frequent updates of logging statements. Zhu et al. [9] presented a comprehensive evaluation study on 13 log parsers on 16 log datasets, including benchmarks of accuracy, robustness, and efficiency. Among the evaluated log parsers, Drain [11] had high average accuracy in 8 datasets, and the highest average accuracy in all datasets. Efficiency was evaluated on 3 datasets with increasing log volumes, and among 6 log parsers, Drain and IPLom had the best efficiency over all tests.

2.4.1 Drain

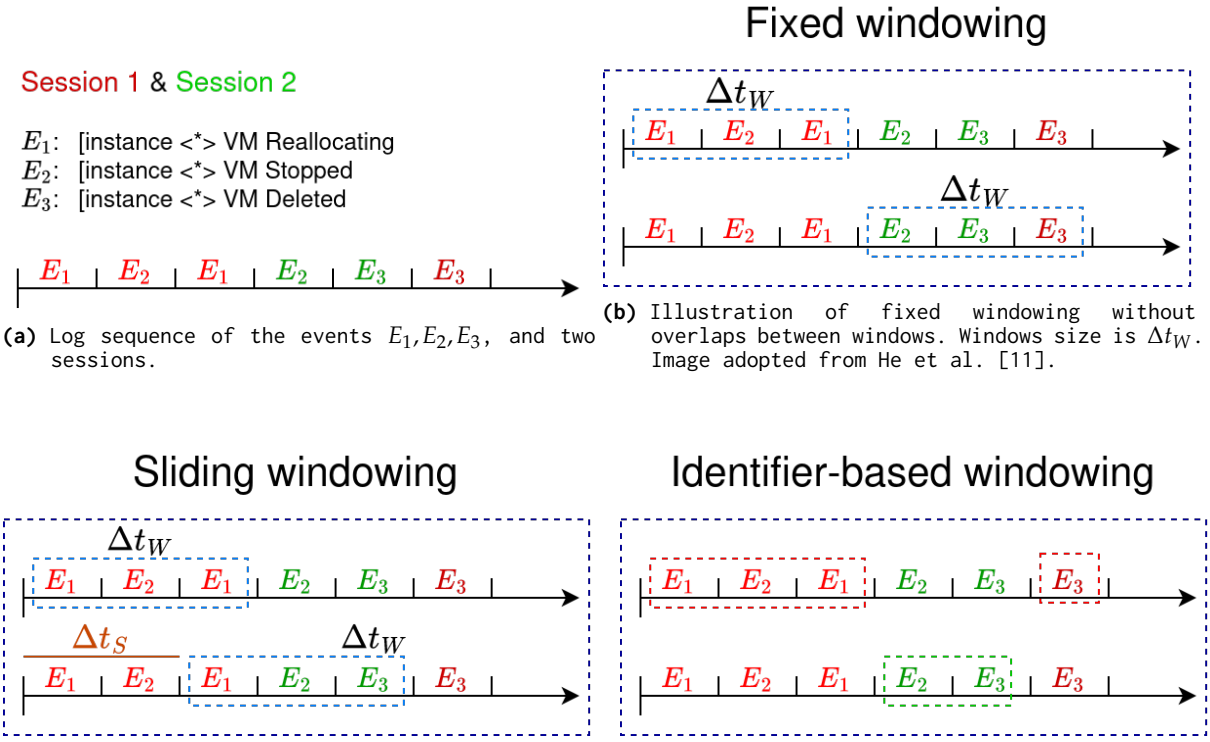
As described in Section 2.4, the Drain [11] log parser showed excellent parsing accuracy and efficiency on a variety of datasets. Furthermore, Drain has open-source availability, online parsing capability, and the latest version Drain3 has persistent storage capability, making Drain a popular choice in the industry, and Lee et al. [12] describes Drain as the most commonly used log parser, and is the chosen log parser in this paper. Drain is designed for accurate and efficient log parsing of raw log messages, automatically extracting log templates and structuring them into disjoint log groups by employing a fixed-depth tree to avoid creating a deep and unbalanced tree. To improve accuracy, users can provide regular expressions based on domain knowledge that Drain employs in a preprocessing step. The tree nodes are described as *root node, internal nodes*, and *leaf nodes*. The root node and the internal nodes encode rules to guide the process of locating a log group for a log message, and the leaf nodes store lists of log groups. If regular expressions are provided, these are initially applied to the log message. The next step is to search by word count of the log message, and then to search by preceding tokens of the log message, based on the fact that constant parts of log messages often occur at the beginning of a log message. The next step is to search by similarity of already parsed log messages, and based on the findings, the log message is added to the most suitable log group. The log event is the template that best describes the log message and the log IDs are identifiers to each processed log message within the log group.

As such, the process can be explained as the following steps:

- Tokens are generalized based on provided regular expressions.
- Split each log message into words (tokens) and build a new fixed-depth tree based on token length if it does not exist, else traverse down the existing one.
- At each tree level, new messages are compared with existing children. If the tree already has *max_children*, it uses hashing into existing nodes, otherwise a new node is added.
- Grouping into existing nodes is done with the parameter *st* (similarity threshold), and a higher threshold require log messages to be more similar before they are merged together.

2.5 Log Partitioning / Windowing

After parsing log messages, log sequences may contain hundreds of thousands of logs, and therefore log sequences are often partitioned into smaller chunks [13]. He et al. [11] categorizes partitioning into *timestamp-based partitioning* and *identifier-based partitioning*; identifier-based partitioning (also called *session windowing*) groups logs with unique identifiers, e.g. the logs originate from the same task execution, and timestamp-based partitioning groups logs into time windows, either into *fixed windows* or *sliding/overlapping* windows.



(a) Log sequence of the events E_1,E_2,E_3, and two sessions.

(b) Illustration of fixed windowing without overlaps between windows. Windows size is Δt_W. Image adopted from He et al. [11].

(a) Illustration of sliding windowing that overlaps between windows. Windows size is Δt_W and step size is Δt_S. Image adopted from He et al. [11].

(b) Illustration of session windowing that only selects those messages relating to each session. Image adopted from He et al. [11].

Figure 2.5: Windows of a log sequence

2.6 Feature Extraction

In the context of log anomaly detection, *feature extraction* is the process of identifying and extracting various characteristics from the data and transforming the data into numerical formats, ready to use in machine learning algorithms. The features in the collected datasets may be domain-specific to each system, and the process may be exploratory and iterative. The features may have strong correlations, and the extraction of some features may exclude other features that should have been included for a more comprehensive understanding of the data. As systems grow in complexity, it is not straightforward to determine whether extracting some parts of the data would lose key information. This section includes the most common feature extraction techniques and their strengths and weaknesses.

Ma et al. [13] categorize features as *graphical* or *digital* features. *Graphical* features process the information from the logs to produce nodes and edges that represent relationships between different log events. Compared to digital features, graphical features are better at describing intrinsic relationships between log events. *Digital features* are the most common feature type, such as values or strings directly extracted from structured logs. The most common digital features are *main features* that do not depend on the variable parts in their corresponding log messages, but to improve accuracy, *secondary features*, represented by their variable parts, can be used alongside the main features in downstream analysis.

Log ID features are sequences of log IDs where each ID represents an identifier for a specific log event, and using Log ID as the extracted feature preserves the positional relationships between log events and reduces the textual size of each message, since each ID is a short string or number. The weakness of the feature is that it loses the textual context information for each log message, and training requires language models to learn sequences of IDs. Chen and Liao [14] showed that it is possible to train models on log ID features for log anomaly detection.

Log event features focus on the text of the log message, to capture the semantic information in the log message and possibly group similar log messages even if they are formulated differently. Log event features are gaining more attention since they are capable of matching unseen log templates to existing templates based on semantic features, making them more robust to changes in the log message formats.

Log count features count the frequency of log events in partitioned log sequences, producing a *log count vector* for each sequence, where each index represents a unique log event and each index value the frequency of the log event. The vectors are compact representations of the sequences, and the longer the sequence, the more information is packed into a single vector, which reduces computational demands during down-stream analysis, but the weakness is that the order of log events is lost. A common representation of multiple log count vectors is to construct log count matrices, where each row corresponds to a log count vector from a specific time interval. There are no limitations on which features to count and can be any information provided with the log message, such as log ID, process ID, severity level (e.g., INFO, WARNING, ERROR), host, or program, etc.

Another method is to extract *temporal features*, allowing to examine patterns that change over time, capturing trends in the logs data, such as the frequency of certain log events over time. A concrete example is to form log count matrices, such that each log count vector represents the same period (e.g. 02:00 - 02:30), but each unique vector represents the time interval for a unique day.

Another feature is to examine *correlations*, which is important to consider before separating data, since the separated data may lose some correlated information between different log messages. Correlations may be computed as correlation matrices, with values indicating how strongly correlated features are to each other, and higher cell values indicate stronger correlations. These correlation matrices can be visualized as heatmaps, as shown in Figure 2.6.

Figure 2.6: Correlations between OpenStack modules during a short period.

2.7 Log Embeddings

The term log embeddings is semantically the same as the term *word embeddings*, but used in the context of log messages. Embeddings are numerical vector representations of words or sentences, and since ML algorithms operate on numerical data as input, the non-numeric variables such as words must be transformed into numbers and vectors.

A traditional embedding method is *one-hot encoding*, where each word is represented by a vector of length equal to the number of vocabulary words, and for each unique word, a specific index has value 1 and the rest of the indices have value 0. Another traditional way of representing words in numerical format is *bag-of-words* (BOW), that groups text into "bags" of words, such that the order of the words is disregarded, but their frequency is taken into account, where the values in each word index represent how many times the word occurs in the "bag". The strengths of *one-hot encoding* and *BOW* are their simplicity and interpretability. The weaknesses are that each word is represented by a large vector if the vocabulary is large, and since each item is technically equidistant in vector space, there is no context similarity between words, and as a result, words with little variance are no closer to each other than those with high variance.

Word2Vec [15] is a more avanced method that may be used to create embeddings. It is a group of NN models trained on a large corpus of text to learn word relationships from the texts. The models create word vectors that capture meanings based on the words' contexts, helping to understand their semantic relationships. Word2Vec uses two model architectures when learning distributed representations of words; *continuous bag-of-words* (CBOW) and *continuous Skip-gram*. CBOW predicts the current word based on the surrounding words, and continuous Skip-gram predicts surrounding words based on each word. Word2Vec encodes words into one-hot vectors and feeds the words into a hidden layer, which generates hidden weights that predict other nearby words, returning the hidden weights as embeddings. A strength

of the Word2Vec models is the underlying shallow neural networks, which leads to shorter training times. A weakness is that each word has the same embedding regardless of its context. Another weakness is that embeddings used in similar contexts have similar embeddings, even if the the words are antonyms (opposite meanings).

A more modern approach is to generate embeddings using LLMs pre-trained on large volumes of text, allowing the embeddings to capture the semantic content of log messages. This is done either directly or after fine-tuning the model on the logs, which often contain domain-specific terminology and sentencing.

2.8 Log Anomaly Detection

In the literature, *anomalies* and *outliers* are sometimes used interchangeably, but *anomalies* is the predominant term. To define an anomaly, the range of normal values must first be established, which involves setting a threshold that an anomalous value must exceed. *Anomaly detection* is the process of applying methods to determine whether the data conforms to expected behavior [16], and there are many different methods, typically separated by *supervised* and *unsupervised* methods. Supervised methods rely on labeled data to learn patterns between normal and anomalous data before they are able to detect anomalies in new data. Unsupervised methods do not need labeled data and learn normal patterns on their own and identify anything that deviates from those patterns as a potential anomaly. This study focuses only on unsupervised methods.

2.8.1 Anomaly Metrics

To assess anomaly detection performance with metrics, the assumption is that each value is either labeled normal or anomalous, with normal values described as negative (N) values and anomalies as positive (P). The process is then to match each prediction with the actual values, and count them as either *TP* (True Positive), *FP* (False Positive), *TN* (True Negative), or *FN* (False Negative). These values are typically visualized using confusion matrices, such as the one in Figure 2.7.

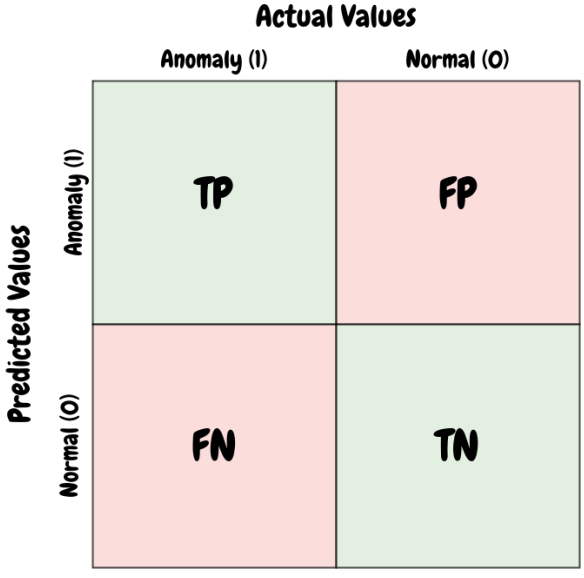


Figure 2.7: Confusion matrix when used in the field of log anomaly detection, showing the desired TP and TN with green background and the unwanted FP and FN with red background.

The optimal situation is that all anomalies are correctly predicted (*TP*), and all normal values correctly predicted (*TN*), but there may be false predictions classified as FN and FP, and to assess the anomaly detection performance in multiple regards, the values in the confusion matrix are used to compute the metrics *Precision*, *Recall*, and *F1-score*. *Precision* is the ratio of correctly predicted anomalies over all predicted anomalies, with a high score indicating that the false positives are minimized. *Recall* is the ratio of correctly predicted anomalies over all anomalies, and a high score means that most anomalies are correctly predicted, without accounting for the number of false positives. *F1-score* is a harmonic mean value that balances both *Precision* and *Recall*, which makes it a suitable metric to show how well a model performs in anomaly detection.

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

It is worth mentioning that another metric is *accuracy*, which means all true predictions over all predictions, but with the assumption that anomalies are scarce, accuracy is not a suitable metric, because a model that always predicts values as normal is going to achieve high accuracy, despite the fact that the model could not detect a single anomaly.

2.8.2 Statistical Methods

Z-Score Analysis is a statistical method that measures standard deviations from the mean value, commonly used to identify outliers in the data. The method has low computational complexity but is sensitive to outliers in the data that affect all computed values, and the method requires normally distributed data. As described by Rousseeuw and Hubert [17], another variant is *Robust Z-Score*, which addresses the limitations of normal Z-score, using the median value instead of the mean value in a computation that creates deviations similar to normally distributed standard deviations.

PCA (Principal Component Analysis) is a statistical method in linear dimensionality reduction. It transforms the data and produces principal components, that represent data with higher variance. The produced components are ranked in descending order, such that the first component captures the largest variation in the data. It is commonly used for dimensionality reduction, but can also be used as an unsupervised log anomaly detection method to identify patterns and structures. In earlier research in log anomaly detection, Xu et al. [18] showed that PCA can achieve decent results in log anomaly detection, by creating feature vectors from log data, applying term-weighing techniques, and employing PCA in the feature vectors, which would separate repeating patterns, making anomalous message patterns easier to detect. In a large set of logs, the authors claimed that PCA is suitable due to the linear runtime to the number of feature vectors. In an experience report on log analysis, He et al. [19] found that PCA is hard to understand and interpret, that it is sensitive to the data, and that anomaly detection performance varies between datasets.

2.8.3 Isolation-Based Methods

Isolation Forest [20] is a traditional method that isolates anomalies in the data by randomly selecting features and then randomly selecting split values between the maximum and minimum values of these features. The algorithm builds a forest from binary trees, where each tree isolates points from the rest of the data with anomalies typically isolated closer to the root due to their distinctiveness. The trees in the forest collectively determine the anomaly scores based on the path lengths to the isolated points, such that shorter paths in these trees indicate anomalies. The complexity of the model mainly depends on the subsampling size n and the number of trees t in the forest, making the worst-case time complexity of training $O(tn^2)$ and the space complexity $O(tn)$.
Liu et al. [21] constructed a log anomaly detection model that utilized Isolation Forest on unlabeled log data. The authors claimed that abnormal logs are typically few and different from normal logs and therefore easier to isolate, making Isolation Forest an applicable choice in anomaly detection. The authors found that when the subsampling size increases to a desired value, Isolation Forest detects reliably and there is no need to increase the size any further, because it increases processing time and memory size without gaining accuracy, and setting the size to 256 works well for a wide range of log data. For their datasets, they found that using 100 trees was more than enough, such that the path lengths usually converge well before 100 trees.

2.8.4 ML Methods

Earlier work in sequential log analysis used Recurrent Neural Networks (RNNs), designed with an internal state that is updated as the network processes input elements to capture information about earlier inputs in the sequence, but for sufficiently long sequences, they create vanishing or exploding gradients.

Long Short-Term Memory (LSTMs) are more complex RNNs, designed to solve vanishing / exploding gradients [22], making them capable of processing longer sequences than traditional RNNs. The central parts are three types of gates that control the information flow in the network, such as which data to memorize, discard, or use as output. LSTMs are commonly used in log anomaly detection due to their capability of handling long sequences of data. Deeplog [23] used an LSTM to learn log patterns from normal executions to detect anomalies whenever the log patterns deviated from the trained patterns.

Convolutional Neural Networks (CNNs) are predominantly used to process data with a grid-like topology, such as images, by employing layers of convolutional filters that apply various transformations to the input data. Lu et al. [24] showed that CNNs can be used in log anomaly detection, by constructing a model that achieved high accuracy on a publicly available HDFS dataset.

Gated Recurrent Units (GRUs) are newer RNNs designed to mitigate the vanishing / exploding gradients. They are smaller architectures with only two gates, thus having fewer parameters than LSTMs, making GRUs suitable when smaller architectures are sufficient [25, 26].

2.8.5 Transformer-Based Methods

Bidirectional Encoder Representations from Transformers (BERT) is a transformed-based LLM developed by Google researchers. It processes input from both directions for a deeper understanding, making the output contextual, so that each word has a different embedding in different contexts. The training is self-supervised, where parts of the input tokens are masked and the model learns to predict the masked tokens during training. This kind of training is called Masked Language Modeling (MLM) and is similar to cloze tests for humans, where certain words are hidden and the task is to predict the missing words based on the context. BERT is pre-trained with MLM on large corpus of texts, but to use it in domain-specific

languages such as log messages it requires additional fine-tuning. The weakness of BERT are the sentence embeddings, typically represented by the first token in a sentence called the [CLS]-token. It may not be the most suitable, described by one of the authors of BERT, because BERT is not designed to generate meaningful sentence embeddings, and comparing [CLS]-token vectors with cosine similarity is not possible, since cosine similarity is a linear space where all dimensions are weighted equally. Guo et al. [27] used BERT in their log anomaly detection method and trained BERT with MLM on template sequences, showing promising results on publicly available log datasets.

Sentence-BERT (SBERT) is a modified BERT architecture built as a siamese network of identical BERT models, specifically designed to generate semantically meaningful sentence embeddings, with initial training on large corpus similar to BERT and then fine-tuned on sentence pairs to learn semantic similaritites. The developers describe how the architecture improves performance in tasks such as semantic similarity assessment and clustering, making the model viable for tasks in log anomaly detection when the log templates (the sentences) are converted into embeddings (the numerical representations). When comparing SBERT with BERT, SBERT produces sentence embeddings that can be directly compared using cosine similarity, making them useful when working with different sets of templates, as the embeddings can be evaluated in combination with actual anomaly detection results. There are several SBERT models to choose from, each with specific strengths and weaknesses such as size, efficiency, and how well they perform on semantic text tasks. According to the model documentation [28], the best performing model is *all-mpnet-base-v2*.

3 Related Work

Chen et al. [29] wrote a survey on log anomaly detection, providing extensive coverage of the topic. The authors mention that log messages are growing in size and complexity, explaining that manual inspection of log messages has become an infeasible approach and there is a need for efficient and accurate ML models to handle the modern size and complexity of log messages. The paper is a comprehensive experience report in the field of log anomaly detection, covering topics such as log collection, log parsing, feature extraction, and anomaly detection. The paper also discusses partitioning the data in time windows and frequency counting of log types as input to the ML models. The paper includes multiple state-of-the-art methods, their strengths and weaknesses, and a comparison between them in terms of anomaly detection accuracy and running time. A key finding in the paper is that reconstruction-based methods are more robust than prediction-based methods when the input data contains anomalies, making them more applicable to real-world datasets without labels.

The next research paper was [9], which is a comprehensive study of log parsing, including details of state-of-the-art log parsers and benchmarks of log parsers in terms of efficiency and accuracy. The study showed that Drain is an efficient and accurate log parser, with several features suitable for this thesis, such as being open-source and capable of persistent storage. He et al. [11] served as the next paper, written by the developers of the parser Drain, covering the overview of the parser and a deep dive into the intricate details of the parser implementation, and explained how the parser can be tweaked to suit specific needs in domain-specific log messages.

Guo et al. [27] was the first paper that introduced this study to transformer-based methods, using BERT in their log anomaly detection model, fine-tuning BERT on log templates with the self-supervised methods Masked Language Modeling (MLM) and Volume Hypersphere Minimization (VHM). The included benchmarks showed decent accuracy results, outperforming the other models that LogBERT was compared to. Further embarking on log anomaly detection with BERT led to BERT-Log [14], which is another BERT-based model, trained with MLM on sequences of log IDs instead of sequences of log templates, and showed that this approach is another viable option and that the model could accurately distinguish normal sequences from anomalous ones. The authors write that after training, extracting the embeddings from the [CLS]-token in the last hidden layer of BERT can represent the semantics of the sentence. Another BERT-based model is LanoBERT [12], which is a parser-free log anomaly detection model that uses BERT, completely discarding the next sequence prediction (NSP) function. LanoBERT is trained with MLM in sequences of log templates, randomly masking 20 % of the templates. The model is inferencing on sequences of log templates, such that each log template in each sequence is masked, stepping forward one position each time. For each masked token, the probability and loss of the most likely token is stored. Then for each sequence, the top k worst probabilities and top k highest losses are averaged, and if the averaged value is higher than some threshold, the sequence is determined anomalous. The paper does not discuss the complexity of the inferencing stage, which is likely high due to the need to step forward each masked token for all tokens in each input.

Autoencoders (AEs) built with NNs were first introduced in 2006 by Hinton and Salakhutdinov [30], and have since become a common ML tool. The technique is practically the same for AEs with deep NNs, which is to propagate the input information through intermediate deep NNs to gradually compress the

dimension into a latent space, and then gradually expand the dimensions through more intermediate deep neural networks back to the original dimension, and during the training, the model learns to reconstruct the data with minimal loss. The anomaly detection is evaluated by how well the model reconstructs the data such that a large reconstruction loss is an indicator of an anomaly.

With the advent of recurrent NNs such as LSTMs, the AEs are able to learn relationships among input data for much longer sequences, which has proven successful in log anomaly detection. Grover [31] used both a prediction-based LSTM network and a reconstruction-based LSTM (autoencoder) with 15-35 neurons in the intermediate LSTM layers and discovered that both methods were decent, but the autoencoder performed slightly better in discovering anomalies and could generalize better. The author analyzed the datasets HDFS and BGL and created log count vectors for each session as model input, making the vector length of each input equal to the number of parsed log events, but the author does not explicitly mention these input sizes. Previous papers that use HDFS and BGL datasets use between 30 and 50 for HDFS and close to 400 for BGL and therefore the initial compression of the LSTM on the HDFS dataset may be minimal and may be significantly larger for the BGL dataset, which may affect the performance of the underlying LSTM networks, and it is also worth noting that their methodology of using log count vectors is not able to detect sequential anomalies in each session. Furthermore, the author does not explain any details of anomaly classification based on the reconstruction loss, only that the anomalies are those with higher reconstruction loss than the loss values on training data.

Utilizing bidirectional LSTMs in AEs is also used in log anomaly detection models, because the neural networks can learn to recognize patterns from both directions of the input. Lee et al. [32] used both unidirectional LSTMs and bidirectional LSTMs in AEs and showed that bidirectional LSTMs improved anomaly detection on four types of datasets. The authors reasoned that the improved results are caused by the input flows in both forward and backward directions, which preserves both past and future information.

Torabi et al. [33] detects anomalies with an AE on the dataset *CIDDS-001*, which is a dataset designed for network-based intrusion detection. The authors used the MAE loss function and set threshold values based on the maximum error loss during training. Furthermore, the authors used a threshold value for each feature in the dataset and claim that this improves the anomaly detection performance. Their model achieved high results compared to the other methods used in their study. It is worth noting that the type of internal NNs is not specified in the paper, nor is the reason of using the MAE loss function instead of the typical MSE.

Raihan and Ahmed [34] detects anomalies with AEs in a wind power dataset. They authors built one unidirectional AE and one bidirectional, and showed that bidirectionality improved the anomaly detection rate. For each AE, both the encoder and the decoder is using two LSTMs and likewise in the decoder, with 64 dimensions in the outer LSTMs and 32 in the inner LSTMs. The threshold for the reconstruction error is based on the distribution of the loss values during training. It is unclear if the decoder in the bidirectional LSTM is bidirectional or not, and it would be interesting to know more details on how the threshold value is determined based on the distribution of training losses.

Hu et al. [35] developed another anomaly detection method that uses a bi-LSTM that trains on SBERT-generated embeddings to learn patterns. The authors mention the use of a self-attention mechanism, but it is unclear how and where this is applied. The authors also mention that the bi-LSTM is using a hidden size of 64, but there is no mentioning of the model being an autoencoder, and as such their method is quite abstract, and it would be interesting to know more details. The authors show promising results on the HDFS dataset when compared to previous methods such as DeepLog and LogAnomaly, and also reveal results from parameter-tuning that may prove useful in this study as guidelines of initial values.

4 Solution Design

This chapter explains how the research questions have been answered and how the model design is formed from the background study.

4.1 Features from Unstructured Data

To address the challenges of extracting meaningful features from unstructured logs, the solution is based on the findings in the background study. As described in Section 2.4, a key part of log analysis is structuring logs through log parsing, classifying logs into groups so that each group represents the same type of message, with a unique log template and log ID. Therefore, before any numerical features can be extracted, the initial step is to parse the logs, and the Drain log parser (described in Section 2.4.1) was chosen for its set of features and its performance in research comparing different log parsers.

The next challenge is to extract meaningful information from the logs, which involves representing the data as numerical features that are relevant for anomaly detection.

For the labeled HDFS dataset, the approach is to use a pre-trained LLM to generate embedding vectors from sentences, where each sentence corresponds to a log template. SBERT is the chosen model because its architecture is designed to produce meaningful embeddings that can be compared using cosine similarity. Another reason for using SBERT is its ability to be fine-tuned with sentence pairs and similarity scores, allowing related sentences or words to be positioned closer or further apart in the embedding space.

For the ERDC dataset, the extracted features are varied and not limited to a predefined set. They include basic statistics such as overall log density across all services or between specific services, as well as more detailed metrics such as log distributions within subservices and cross-service patterns over different time intervals. This broad selection of characteristics is intentional because the root cause of the issue is unknown and requires examining a wide range of patterns. Additionally, each log message contains many structured fields, such as severity level, process ID, thread ID, OpenStack module, program name, Python module, hostname, etc. Some of these fields may be empty, and deciding which ones to include has been an exploratory process.

For the voluminous ERDC dataset an impractical approach is to use the entire dataset for analysis based on the large-scale of the dataset, comprising more than 500 million log messages. As described in Section 2.5, logs can be partitioned into windows to count the frequency of different characteristics, which has the benefit of reducing the size of the log data and reducing noise for sufficiently large windows. As described in Section 2.1, another reason for creating log count vectors is that the ordering of OpenStack logs may be interleaved in OpenStack, due to the system's design with fairly isolated services and possible asynchronous operations both within the services and between them. The weakness of using log count vectors is that the message order is lost, and therefore this method is applied as a course-grained method to get an overview of the data. When anomalous windows have been identified, log partitioning is applied with shorter window size, and as described in Section 2.5, sliding windows is an overlapping windowing method, which has the benefit of capturing more information about the logs than fixed windowing, avoiding

missing potential information between fixed windows.

The start and end times of the logs vary slightly from day to day, and to account for these differences, the smallest time window is set to five minutes, which aligns with the OpenStack log rotation cycle across all days. To simplify how the windowing is done on the ERDC logs, all window sizes and step sizes are factors of five minutes. Continuing on windowing, it is worth noting that it is a difficult concept and there is no one size fits all. Thus, establishing the window sizes and step sizes in this study has been an exploratory process. The main theme when tailoring the windows has been to initially apply larger windows to get an overview of the logs, and to reduce the windows for a detailed view, thus forming an iterative process to narrow down which data to select for further analysis. Another design choice worth mentioning is to include an extra window at the end of the sequence if the last window does not align with the final entry in the sequence. The reason is to account for missing values at the end of the sequence, used for both fixed windows and sliding windows, possibly including an overlapping pair of windows in fixed windows and a different ending step size in sliding windows. Windowing in combination with the chosen log categories is used to create log count vectors, counting the frequency of each chosen category in each window to form log count matrices from each set of log vectors. These log count matrices are used as the foundation for selecting more isolated and detailed data to be used in a fine-grained analysis with deep learning.

4.2 Handling Unlabeled Data

Without labeled data, the proposed design relies on unsupervised learning methods, and the main theme of the solution is to incorporate efficient methods during the course-grained analysis of log count matrices to locate smaller log subsets to use for deep learning methods.

For the course-grained analysis, most log count matrices have limited size and are not suitable for larger models such as the autoencoder (AE), that require extensive training to learn patterns. As described in Section 2.8.3, Isolation Forest is an efficient and unsupervised method designed to efficiently and accurately isolate the features in data, and is therefore chosen to analyze the log count matrices built from the ERDC dataset. For each matrix, Isolation Forest isolates anomalies and produces anomaly scores for each window (each matrix row), such that larger scores are more likely to be anomalies. To reduce noise, a threshold is applied on the anomaly scores, such that those scores below the threshold are set to zero. As described in Section 2.8.2, Z-score is a simple statistical measure that represents the number of standard deviations from the mean value. Since this metric requires normally distributed data and the fact large anomalies affects other data, Robust Z-score is instead used, which uses the median value instead along with a constant that scales to imitate normally distributed standard deviations. With the use of Z-scores and Isolation Forest anomaly scores, different categories can be compared, and time intervals can be compared in an anomaly ranking process. For sequential analysis of embeddings, the unsupervised approach is to design an autoencoder (AE) that trains on embedding matrices and learns to reconstruct the data. The trained AE is then applied to other embedding matrices, and for each matrix, a reconstruction loss is obtained, which represents how well the model is able to reconstruct the data, and large reconstruction losses are treated as anomalies. In this study, the initial threshold for reconstruction loss is set to the average loss during training, but it is further fine-tuned by testing a range of standard deviations around that average.

Continuing on the AE, there are nearly endless design options, such as the underlying NNs, number of NNs corresponding to compressions and expansions, size of each compression and expansion, number of stacked NNs in each compression and expansion, dropout value of neurons whenever stacked NNs

are used, bidirectionality of NNs, and much more. To limit the design options to a reasonable amount, parameter findings by Hu et al. [35] is used as a starting point, where they designed a bi-LSTM to train on SBERT embeddings, similar to how the AE in this study is trained. The authors reveal that using two stacked layers in their bi-LSTM significantly improves model performance, and that a hidden size of 64 significantly improves performance compared to 32, while further increasing the hidden size has almost no effect. The authors also show that for the HDFS dataset, using window sizes of ten logs is sufficient to achieve optimal results. With guidance of the results from their paper, this study uses their fine-tuning results as inspiration for determining initial values and to narrow down which parameters to include in the tuning process. Therefore, during AE fine-tuning in this study, a window size of ten logs with a sliding step of five logs is used to capture information that might be missed in fixed windows. The number of compression layers is limited to a maximum of two, and for simplicity, the smallest compressed size is always set to 64. To maintain a symmetric design, the number of expansion layers matches the number of compression layers, and to avoid other parameters, the dropout value is always zero.

To separate AE fine-tuning from reconstruction loss threshold fine-tuning, a range of threshold values is generated among all possible loss values, and the anomaly detection scores are examined for each of the generated thresholds.

4.3 Datasets

To understand the datasets, this section provides details about them and how they are used in this study.

HDFS is a publicly available dataset, which is labeled, chosen based on its popularity in research. It is the HDFS_v1 [36] dataset from Loghub, which consists of log sequences collected from the Hadoop Distributed File System [37]. As recommended by the dataset providers, the following papers are cited [18] [38]. The logs can be grouped into session windows, grouped with block ID. As such, each block has an isolated log sequence corresponding to its unique block ID. There are 575,061 blocks with 16,838 of them labeled anomalous. 80% of the normal sequences are used in training, and the remaining normal and anomalous sequences are used in validation.

ERDC logs are OpenStack logs, collected during 13 consecutive days. In total, there are more than 500 million logs, which can be grouped into various categories such as *service*, *log level*, *program*, etc. Not counting timestamps, there are 13 main categories available for log grouping.

Table 4.1 summarizes them and how they contribute to this study.

Name	Labeled	Purpose
HDFS	Yes	*Benchmarking dataset*, used as a sanity check to assess if the designed AE is able to detect anomalies, and also used to fine-tune the model parameters.
ERDC	No	*Real-world dataset*, used as the final target for the model. This dataset is examined using a wide range of characteristics, to get a better understanding of the datasets and to uncover anomalous patterns.

Table 4.1: The datasets used in this study.

4.4 Criteria for Success

As described in Section 2.8.1, anomaly metrics are *Precision*, *Recall*, and *F1-Score*, which are used in the labeled HDFS dataset to evaluate how the proposed model is able to detect anomalies. These metrics are then compared with a couple of state-of-the-art methods found in research papers. The ERDC dataset does not have labels, so common evaluation metrics cannot be applied. Instead, the results are discussed in collaboration with the external company. Identifying anomalous events that span multiple services within a short time interval is of particular interest, as it may point to a broader anomaly in the system.

4.5 High-Level Design

The initial step is to parse the datasets with Drain, with different sets of regular expressions for each dataset to match the domain specifics of each dataset. The HDFS dataset is then split into training and evaluation dataset as input to an *Embedding Module*. The ERDC dataset undergoes further processing and evaluation in a *Course-grained analysis* before proceeding with sequential analysis. Figure 4.1 illustrates a high-level overview of the design.

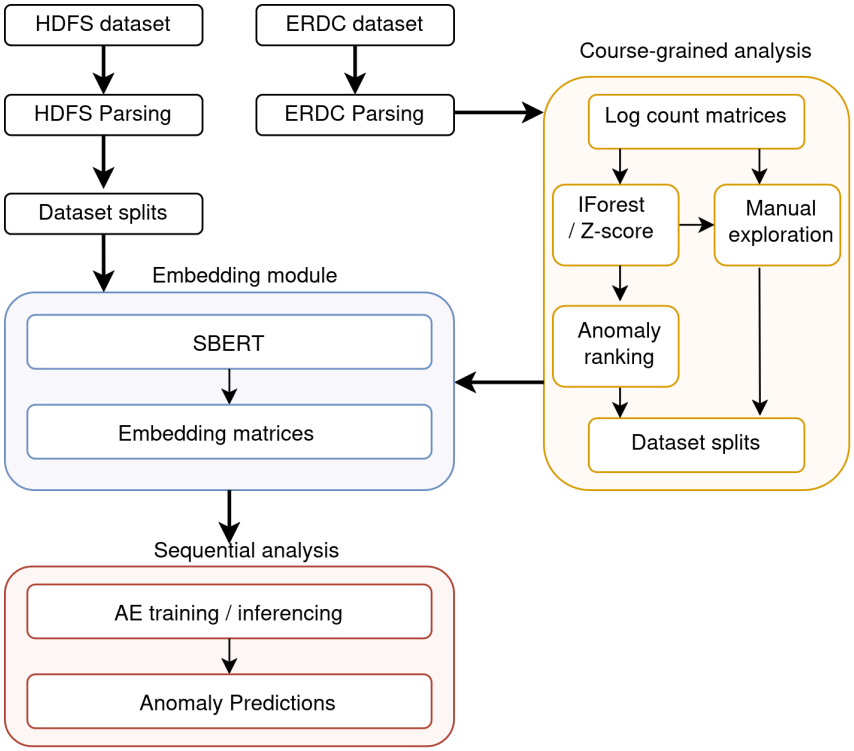


Figure 4.1: Overview of the high-level design

4.6 Course-Grained Analysis

The main purpose of this analysis is to identify anomalous time windows and log count features from the large-scale ERDC dataset. Log sequences extracted from these selected windows and features are then used as input for sequential analysis. The dataset is examined across a broad set of characteristics, primarily by counting log messages over varying time intervals to construct log count matrices. These matrices are then analyzed using Isolation Forest and Z-score methods, each producing anomaly scores for every time window in the matrix. The top 5 % of windows with the highest anomaly scores are selected for sequential analysis.

4.7 Embedding Module

The purpose of this module is to generate embeddings for each corresponding log message using SBERT. These embeddings are then stacked to form embedding matrices, which serve as input for the sequential analysis performed by the autoencoder (AE). In this study SBERT is used without fine-tuning, and the produced embeddings have 768 dimensions, so each embedding matrix with n rows have size n x 768.

4.8 Sequential Analysis

The sequential analysis is done with the designed autoencoder (AE). The AE is initially trained and tested on the labeled HDFS dataset, which enables tinkering with various AE design features and benchmark the AE performance on this dataset. This also allows to examine various threshold values and how these affect anomaly detection performance.

After completing model and threshold fine-tuning on the HDFS dataset, no further adjustments are made. The model is then ready to be applied to the real-world ERDC dataset, where it can be trained on sequences and used to assess reconstruction loss on new data. Any reconstruction loss that exceeds the threshold is going to be flagged as an anomaly.

5 Implementation

This chapter presents how the Solution Design is built, covering the parts log parsing, feature extraction, partitioning (windowing), and methods applied on extracted features. It covers the tools and frameworks used, and how each part of the design was put together, to show how the concepts of the Solution Design were turned into a working setup that can process logs and detect anomalies.

5.1 Tools and Frameworks

All coding is done remotely by logging into the Wara-Ops portal to access any available JupyterHub server installed with the JupyterLab user interface. Everything in this study is implemented with the Python programming language, either as Python modules or as Jupyter Notebooks, using many Python modules, some notable ones shown in Table5.1

Module	Description
pandas	Data representation and data manipulation.
torch	Predominant module for ML-related coding.
pyod	Many anomaly detection models, such as Isolation Forest.
seaborn/matplotlib	Visualizing the data.
sklearn	Mainly used for anomaly detection metrics.
sentence-transformers	LLM module for the model SBERT.
transformers	Tokenizing data along with LLM utility tools.
drain3	Log parsing.
numpy	Efficient numerical computing.
regex	Regex matching.
shutil/gzip	Compressed data storage.

Table 5.1: Key Python modules used in this study.

5.2 Log Parsing

Log parsing is performed with the *Drain* log parser using the *drain3* module. The parser in instantiated with the parameters *similarity threshold* and *depth* along with a list of *regular expressions*. The main focus has been to create domain-specific regular expressions for each dataset. Also, preparing ERDC logs with regular expression matching is done with the *regex* module, since it has more advanced features than the commonly used *re* module, such as variable length look-behinds.

5.3 Log Manipulation

Logs may contain empty values for certain categories, and these are called NaN values in *pandas* tools, which require to explicitly state that the NaN values should be included, and some of these tools do not work with NaN values. Therefore, to always have NaN values enabled in all functions and as uniform formats, these NaN values are transformed to strings “none”.

Log severity levels exist in many formats in the ERDC logs, and therefore similar ones are merged into uniform formats, done with a mapping of the first three letters of each log severity level, and each log severity level is applied to a function that sets the existing log level in lowercase and then converts it to a uniform value.

5.4 Feature Extraction

Each log file is stored as a *csv*-file, loaded into memory as a *Pandas* dataframe, which can be described as a large table, where each row represents a log event and the columns as various categories, such as timestamp, service name, log severity level, programname, process ID, log ID, payload, template, etc. The log files are large and require too much memory when using all the categories, causing the kernel to crash, and the solution has been to load all log files with only a subset of the categories, using the parameter *usecols* in the *read_csv* function.

As described in Section 4.1, log count vectors use five minute intervals to account for various start and end times between days. This was initially accomplished by converting timestamp strings to *datetime* instances and reindexing the dataframe to use these instances as table indices, instead of the default zero-indexing of the rows. However, the process of converting each timestamp string to a datetime instance is time-consuming for large files, and ERDC log files also have unix timestamps, not requiring any value conversions, which significantly reduced the partitioning time. Each partition is a dataframe, and the count vectors are constructed with the *value_counts* function along with index as the start datetime of the partition, and to account for the values present in some partitions but not in others, the *reindex* function is used, filling missing values with zeros. The reindexing is necessary in the analysis, but even more important is to wisely choose the reindexing - OpenStack services (e.g. Nova or Neutron) may have overlapping program names and Python modules but also specific ones only present in certain services, and therefore, mappings between services and other category values are created before log partitioning. Since building these mappings requires reading all rows from all files, they are generated once and saved as *json* files.

As previously described, the shortest time intervals (windows) of log count vectors is five minutes, and the process of enlarging them involves converting the index to *datetime* instances and then applying the *resample* function with the enlarged window size. The five minute log count matrices are generated once and stored as compressed *csv* files, speeding up the process of constructing log count matrices of various window sizes.

The sequential feature extraction is done with the *sentence-transformers* module, creating an instance *SBERT* with the underlying *all-mpnet-base-v2* model, producing embeddings with the *encode* method.

5.5 Anomaly Detection

The AE model was originally designed using the *keras* module, which requires less boilerplate code. However, due to compatibility issues with the remote server infrastructure, the model was instead implemented with the *torch* module as a subclass of *torch.nn.Module*. As described in Section 4.2, the AE is symmetrically designed in the encoder and decoder with an equal number of LSTMs and with an equal compression and expansion rate. The LSTMs in the encoder and decoder are instances of *torch.nn.LSTM*, designed with only a limited set of parameters to tune, such as *compression*, *direction* and LSTM *num_layers* stacked in each LSTM. In this study, the input size is always 768 corresponding to the SBERT embedding dimension, which further reduces the number of parameters to tune. The compression parameters are automatically filled in the expansion parameters in the decoder. Moreover, the output of bidirectional LSTMs have double output sizes and are automatically adjusted during model instantiation. To account for various data sizes, the model is padding and unpacking input data, using *pad_packed_sequence* and *pack_padded_sequence* from the *torch.nn.utils.rnn* module, extending the input data with zeros, which are ignored during training.

Isolation Forest is implemented using the *IForest* model from the *pyod* module. The number of trees is set to *200*, which is doubling the default value of the parameter, accounting for larger and more complex datasets. When the model is applied on log count matrices, an anomaly score is assigned to each row with the use of the *desicion_scores* function. To ensure that *max_samples* does not exceed the actual number of samples (number of matrix rows), it is set to half the total number of samples.

The metrics *Precision*, *Recall*, and *F1-Score* are computed with the *sk-learn.metrics* module, and these functions require predictions and labels as integers, with zeros as normal values and ones as anomalies, and therefore all produced labels and predictions are integers.

6 Testing / Results

This chapter describes the testing procedures and presents the results of the experiments, including testing hardware, model evaluations, data preparation, configuration settings, and how the output was collected. Then each corresponding result is shown to show how the models performed under different conditions.

6.1 Test Environment

Each experiment is written in a Jupyter Notebook and runs on an ERDC server with access to 200 GB hard disk storage and one NVIDIA A100 Tensor Core GPU, and all model training and model inferencing is using the available GPU. Both the input and the collected output is stored on the running ERDC servers.

6.2 Test Coverage

For the HDFS dataset, all logs are used in a random 80/20 split, but with limited test coverage, since the Solution Design is to limit the number of fine-tuning parameters to a reasonable amount. The constant parameters are based on results from previous studies, such as initial window sizes and model compression dimensions.

For the ERDC dataset, the test coverage is mainly the inclusion of the categories *service*, *log level*, *program name*, *python name*, and a combination of these, as these categories represent fairly high-level components that can be analyzed between various users and hosts. A more comprehensive analysis that includes all possible categories and their combinations would not be feasible within the available time. Sequential analysis only includes *user ID* and *request ID* to reduce noise in each sequence, making them more likely to be used as isolated sequences.

6.3 Test Methodology

The first step is to parse the HDFS dataset in multiple iterations to obtain a reasonable set of templates. Once parsed, the next step is to prepare and use the HDFS dataset to benchmark the anomaly detection performance of the proposed AE, which involves parameter fine-tuning and running multiple tests in iterations. The optimal parameters along with the optimal threshold is later going to be used when the AE is applied on the ERDC dataset.

Once optimized, the model performance on the HDFS dataset is compared with other research in terms of *Precision*, *Recall*, and *F1-score*, with F1-score as the main metric that balances both Precision and Recall.

The next step is to analyze the ERDC dataset, which involves a wide range of experiments designed to identify anomalous time intervals, both across the entire system and within each service. Initially, average

payload lengths are analyzed to ensure that nothing unusual is already present in the log files. Then, the experiments proceed to analyze the number of messages in time intervals, starting with the entire system, and across each service.

Then the experiments go deeper in each service, to count messages produced by subcategories such as *programs*, *python modules*, and *log severity levels* using varying time intervals. Each log count matrix is analyzed with Isolation Forest, and the simplest matrices are analyzed with *Z-score* to produce anomaly values, used to compare services in the time intervals. A threshold value is applied on the results from Isolation Forest and Z-score to filter out low values and remove noise in the results, making the remaining values more likely to show anomalies in the data. Both methods produce anomaly scores for each time interval, which are visualized individually for each service as well as summarized across all services.

The results from these ERDC tests are used to identify which data to use in sequential analysis using the AE. The windows with the highest anomaly scores are used as evaluation datasets, and low-scoring windows are used as training datasets. The sequential testing of log templates is designed to group log sequences with *user ID* and *request ID* and train the proposed AE on the log templates corresponding to each log message to identify sequences with unusually high reconstruction loss values and display these values along with the corresponding time intervals.

6.4 HDFS

This section includes results from the entire pipeline when working with the HDFS dataset, ranging from parsing, to feature extraction, to model fine-tuning and threshold fine-tuning, and finally anomaly detection with the established threshold.

6.4.1 HDFS Parsing

The first step is to parse the HDFS dataset, done in multiple iterations, with the aim to mimic the LogHub HDFS templates [39] as close as possible, with the final set of regular expressions shown in Figure 6.1 along with the resulting templates shown in Table 6.1.

```
#Loghub HDFS does not use repetitive address placeholders, captured by the following:
r'(?<=^|[\s,])((?:\d+\.\d+\.\d+\.\d+:\d+)(?:[\s,]+(?:\d+\.\d+\.\d+\.\d+:\d+))*)(?=$|[\s,])'

r'java\..*'
r'blk_-?\d+'
r'\d+\.\d+\.\d+\.\d+:\d+'
r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
r'\b[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9{4}-[a-fA-F0-9]{12}\b',
r'/([a-zA-Z0-9_\-./]+)'
r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
r'\b(?:[a-z0-9](?:[a-z0-9-]{0,61}[a-z0-9])?\.)+[a-z0-9][az0-9-]{0,61}[a-z0-9]\b',
r'\b[a-fA-F0-9]+\b'
r'\b\d{3}\b'
r'\b(?:[A-Fa-f0-9]{2}[:.-]){5}[A-Fa-f0-9]{2}\b'
r'\b\d+[KMG]?B\b'
r'^dfs\.[^\s]+'
```

Figure 6.1: The regular expressions used in the last HDFS parsing iteration.

ID	Template
E1	<*> Adding an already existing block <*>
E2	<*> Verification succeeded for <*>
E3	<*> <*> Served block <*> to /<*>
E4	<*> <*>:Got exception while serving <*> to /<*>:
E5	<*> Receiving block <*> src: /<*> dest: /<*>
E6	<*> Received block <*> src: /<*> dest: /<*> of size <*>
E7	<*> writeBlock <*> received exception <*>
E8	<*> PacketResponder <*> for block <*> Interrupted.
E9	<*> Received block <*> of size <*> from /<*>
E10	<*> PacketResponder <*> <*> Exception <*>
E11	<*> PacketResponder <*> for block <*> terminating
E12	<*> <*>:Exception writing block <*> to mirror <*>
E13	<*> Receiving empty packet for block <*>
E14	<*> Exception in receiveBlock for block <*> <*>
E15	<*> Changing block file offset of block <*> from <*> to <*> meta file offset to <*>
E16	<*> <*>:Transmitted block <*> to /<*>
E17	<*> <*>:Failed to transfer <*> to <*> got <*>
E18	<*> <*> Starting thread to transfer block <*> to <*>
E19	<*> Reopen Block <*>
E20	<*> Unexpected error trying to delete block <*>. BlockInfo not found in volumeMap.
E21	<*> Deleting block <*> file <*><*>
E22	<*> BLOCK* NameSystem.allocateBlock: <*> <*>
E23	<*> BLOCK* NameSystem.delete: <*> is <*> to invalidSet of <*>
E24	<*> BLOCK* Removing block <*> from neededReplications as it does not belong to any file.
E25	<*> BLOCK* ask <*> to replicate <*> to datanode(s) <*>
E26	<*> BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is <*> to <*> size <*>
E27	<*> BLOCK* NameSystem.addStoredBlock: Redundant addStoredBlock request received for <*> on <*> size <*>
E28	<*> BLOCK* NameSystem.addStoredBlock: addStoredBlock request received for <*> on <*> size <*> But it does not belong to any file.
E29	<*> PendingReplicationMonitor timed out block <*>

Table 6.1: The resulting templates from the last HDFS parsing iteration. During manual inspection, these templates resemble the ones produced by LogHub.

6.4.2 HDFS Embeddings

The log templates use wildcards as placeholders and the embeddings for the corresponding templates are generated without SBERT training and nothing is added to the default vocabulary of SBERT. The architecture of SBERT enables pairwise comparison of sentence embeddings with cosine similarity, which shows how they relate to each other in the embedding space, shown in Figure 6.2 with a heatmap, with each identifier as the corresponding log ID (see Table 6.1).

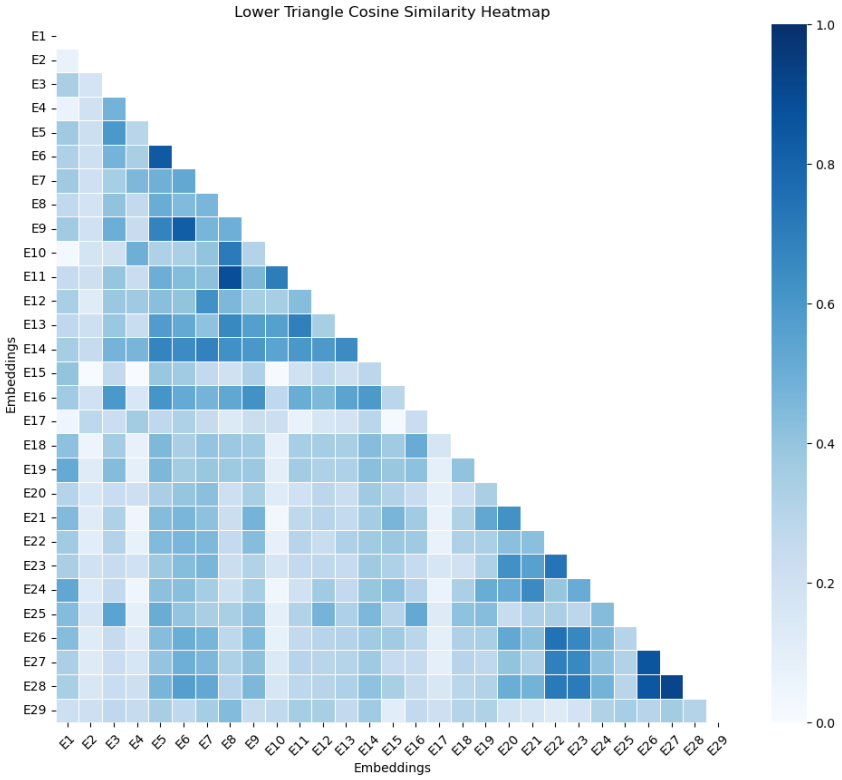
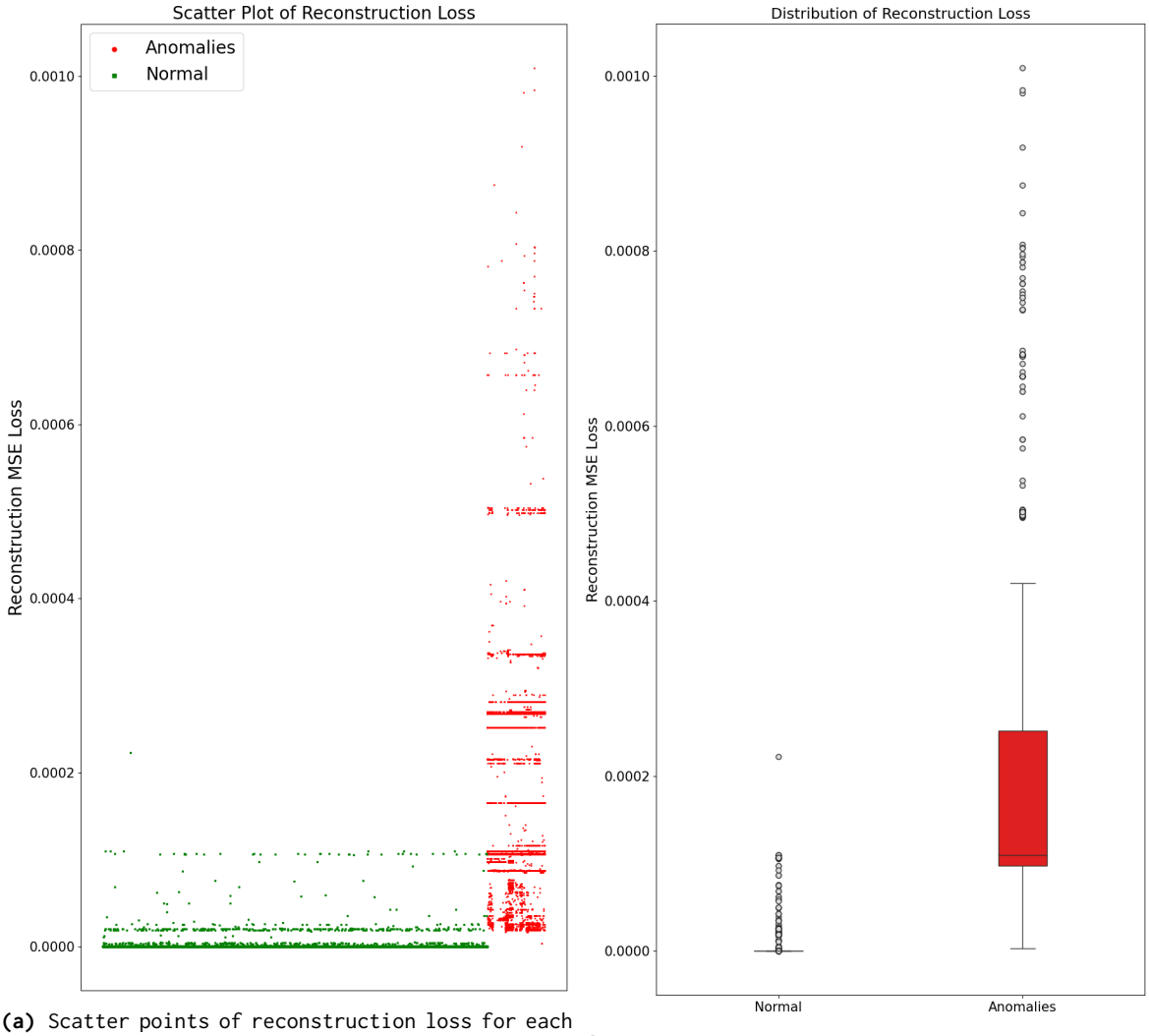


Figure 6.2: SBERT is designed to produce meaningful sentence embeddings that can be compared with cosine similarity. Therefore, it is interesting to see how they relate to each other in the embedding space. In the image, dark cells represent close connections between the embeddings.

6.4.3 Initial AE Testing

The HDFS dataset is separated into a random 80/20 split, such that the 80 % training data contains only normal sequences, and the 20 % evaluation data with the remaining normal sequences and anomalous sequences. The proposed AE is trained on the training dataset to reconstruct the log sequences, and then applied on the evaluation dataset to produce reconstruction loss values. The trained model was bidirectional using the compression 768→128→64 with one-layered LSTMs. Figure 6.3a shows a scatter plot of reconstruction loss values for both normal and anomalous sequences, with green dots for the normal sequences and red dots for the anomalous sequences. A scatter plot allows to view an overall pattern, but it may be challenging to interpret the distribution of loss values for normal and anomalous sequences. Therefore, Figure 6.3b is showing the distributions as boxplots, where most of the normal logs have lower loss values than the anomalous logs, but there exists a range of loss values covering both normal and anomalous sequences. This shows that the model has been able to learn patterns from the normal logs and struggle to reconstruct anomalous logs.



(a) Scatter points of reconstruction loss for each sequence. Green dots are normal sequences and red dots the anomalous sequences.

(b) Boxplots showing loss distributions between normal logs and anomalous logs.

Figure 6.3: Loss values between normal and anomalous log sequences when the trained AE produce loss values on the evaluation dataset. Most anomalous log sequences have higher reconstruction loss values, but there exists a range of overlapping loss values comprising both normal and anomalous loss values.

Prior to establishing a threshold, a set of threshold values are generated between the smallest and the largest reconstruction loss. Then for each threshold, anomaly detection is performed, such that normal loss values are those below the threshold. Figure 6.4 shows how the anomaly detection rate changes for various thresholds (error bounds). Since most normal log sequences have low reconstruction loss values, it is sufficient to set a low threshold value. As the threshold increases, less anomalous sequences are treated as anomalies, shown with decreasing *Recall* and *F1-score*.

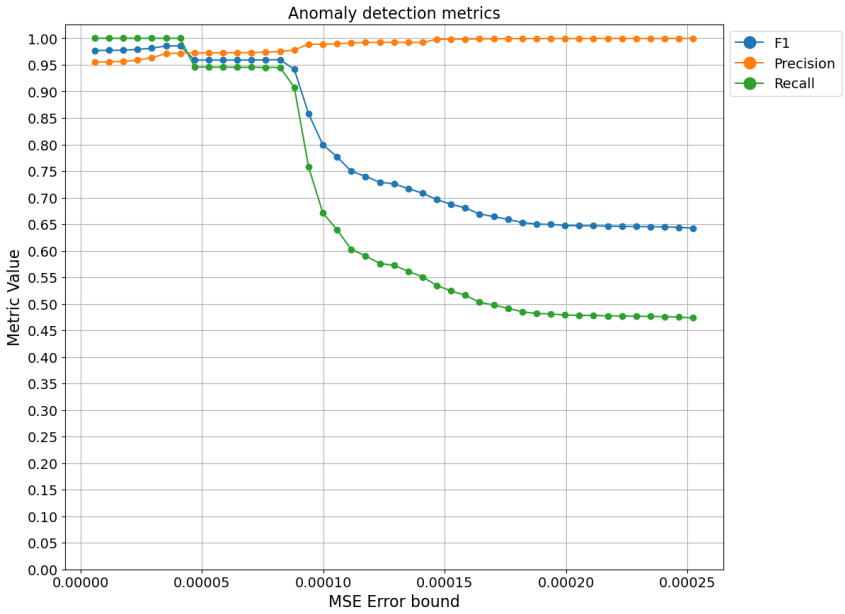


Figure 6.4: Overview of anomaly detection before setting a specific threshold. Instead, threshold values are generated between the smallest and largest reconstruction loss, which allows to assess the anomaly detection rate for a range of threshold values. For sufficiently large threshold values, more anomalies are missed which decreases Recall and therefore also decreases F1-score.

6.4.4 AE Threshold Fine-Tuning

Establishing a threshold value based on the reconstruction training losses is done by constructing a set of AE models, each with a unique set of parameters, and performing anomaly detection with generated thresholds. Then different kinds of thresholds are used to assess if they may be applicable. Using the average loss plus two standard deviations worked across all tested models during threshold fine-tuning, and was chosen despite the data not being entirely normally distributed. Figure 6.5 shows the use of average loss value plus standard deviations on one of the models. Other types of thresholds and how they affect anomaly detection on a variety of models can be seen in Appendix A.

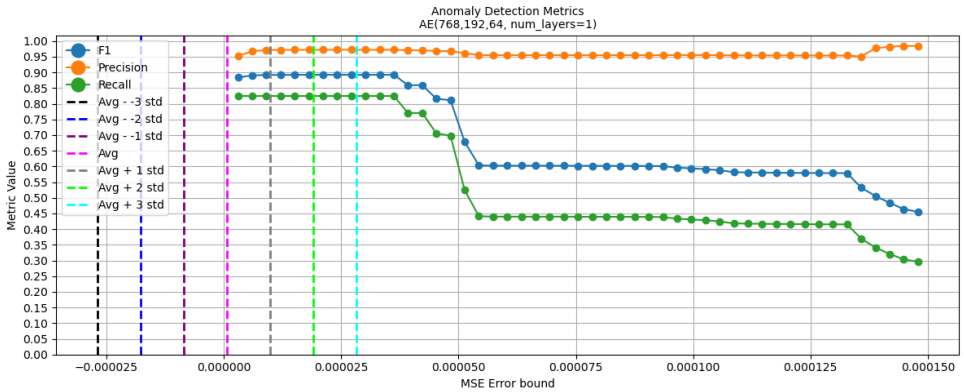


Figure 6.5: Unidirectional AE using one-layered LSTMs with compression 768→192→64. Tested thresholds are average loss plus standard deviations.

To evaluate how different window sizes affect anomaly detection in regard to threshold values, one of the models was tested. For each window size, the model was used to compute reconstruction losses along with a range of threshold values, allowing to identify a threshold that works across the window sizes. The result is shown in Figure 6.6.

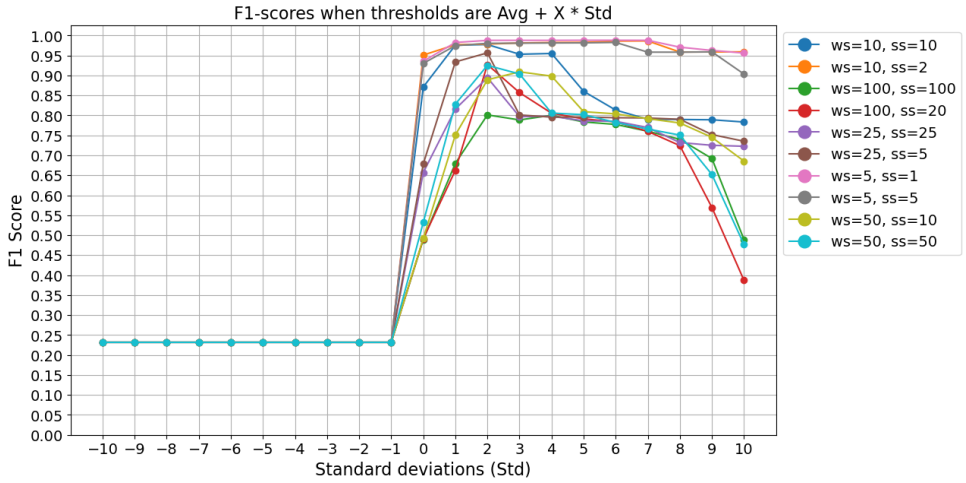


Figure 6.6: Testing a range of window sizes in combination with a range of thresholds. The model is a bidirectional AE with two layered LSTMs using the compression 768→128→64. Over all partitions, the optimal threshold is average loss plus two standard deviations.

Table 6.2 shows anomaly detection metrics across all tested models when the threshold is set to the average loss plus two standard deviations.

Fine-tuned AE	**Precision**	**Recall**	**F1-score**
bi-AE(768,128,64, num_layers 1)	0.973070	1.000000	0.986351
bi-AE(768,192,64, num_layers 1)	0.975890	1.000000	**0.987798**
bi-AE(768,256,64, num_layers 1)	0.973689	1.000000	0.986669
bi-AE(768,320,64, num_layers 1)	0.972957	1.000000	0.986293
bi-AE(768,128,64, num_layers 2)	0.954860	1.000000	0.976909
bi-AE(768,192,64, num_layers 2)	0.954752	1.000000	0.976852
bi-AE(768,256,64, num_layers 2)	0.955239	1.000000	0.977107
bi-AE(768,320,64, num_layers 2)	0.954319	1.000000	0.976625
AE(768,128,64, num_layers 1)	0.972209	0.824801	0.892459
AE(768,192,64, num_layers 1)	0.972209	0.824801	0.892459
AE(768,256,64, num_layers 1)	0.972209	0.824801	0.892459
AE(768,320,64, num_layers 1)	0.972073	0.824801	0.892402
AE(768,128,64, num_layers 2)	0.970103	0.824801	0.891571
AE(768,192,64, num_layers 2)	0.970307	0.824801	0.891657
AE(768,256,64, num_layers 2)	0.962773	0.824801	0.888462
AE(768,320,64, num_layers 2)	0.946436	0.824801	0.881442

Table 6.2: Anomaly detection across the fine-tuned models. Bidirectionality seems to significantly improve Recall, and using one layer seems to improve Precision. The best performing model is the one-layered bidirectional model with 192 as intermediate compression size.

6.4.5 Comparison with Other Research

The best performing model is compared with other research found during this study. The metrics used for comparison are taken directly from the results reported in other research. Table 6.3 lists the anomaly detection metrics, showing that the proposed model achieves decent results with an F1-score of 0.988.

Method	Precision	Recall	F1-score
BERT-Log	0.996	0.996	**0.993**
LogRobust	0.98	**1**	0.99
LogADSBERT	0.986	0.989	0.988
Proposed model	0.976	**1**	0.988
NeuralLog	0.96	**1**	0.98
LogPrompt	**0.997**	0.947	0.971
LAnoBERT	-	-	0.965
DeepLog	0.96	0.96	0.96
Prog-BERT-LSTM	0.944	0.761	0.843
LogBERT	0.87	0.781	0.823

Table 6.3: Comparing the proposed model with other research in terms of Precision, Recall, and F1-score.

6.5 ERDC

The first tests analyzed the number of logs each day along with the average payload lengths, to ensure that larger log files is not caused by longer log messages. The last two days generate most logs, while the last three days show larger average payload lengths, with the penultimate day having significantly larger payload lengths.

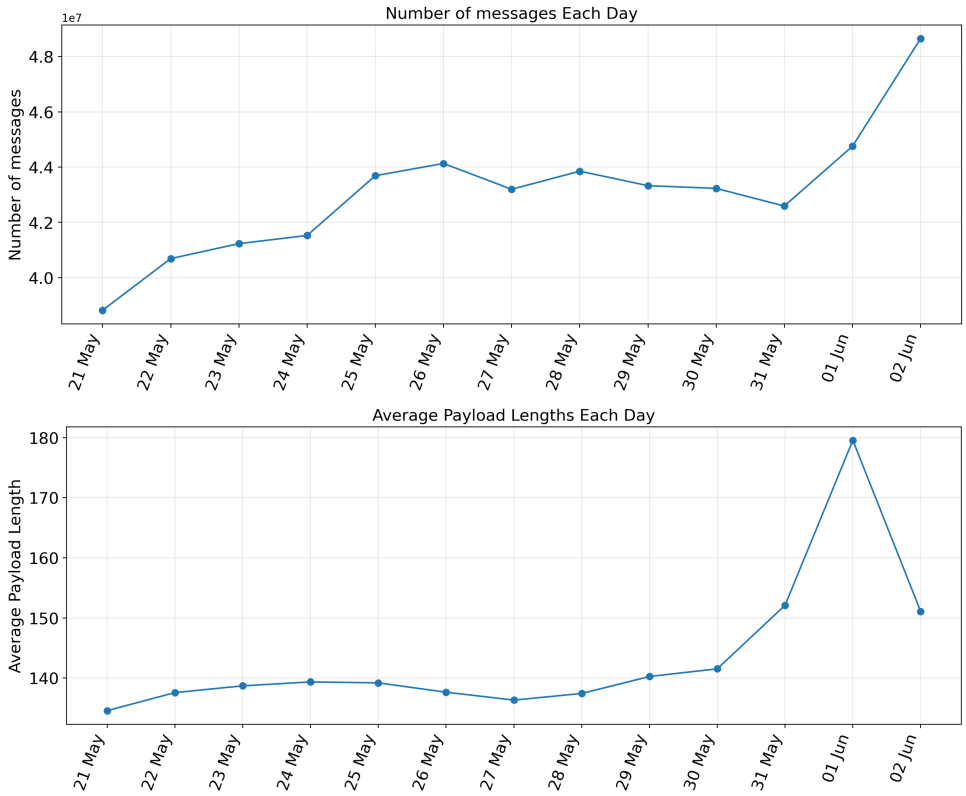


Figure 6.7: To ensure that unusually large payloads are not contributing to the increased log file sizes, the payload lengths are analyzed. Interestingly, the average payload length is higher during the last three days, with 1 June showing significantly larger payloads compared to the other days.

The next tests involved counting logs across all days within each of the services. Using hourly intervals reduced sufficient amount of noise to reveal overall patterns, such as the general amount of logs and periodic daily spikes. Some services have significantly more logs during the last few days, while other services seem largely unaffected. Figure 6.8 shows hourly logs in the most affected services. Less affected services are shown in Appendix C.

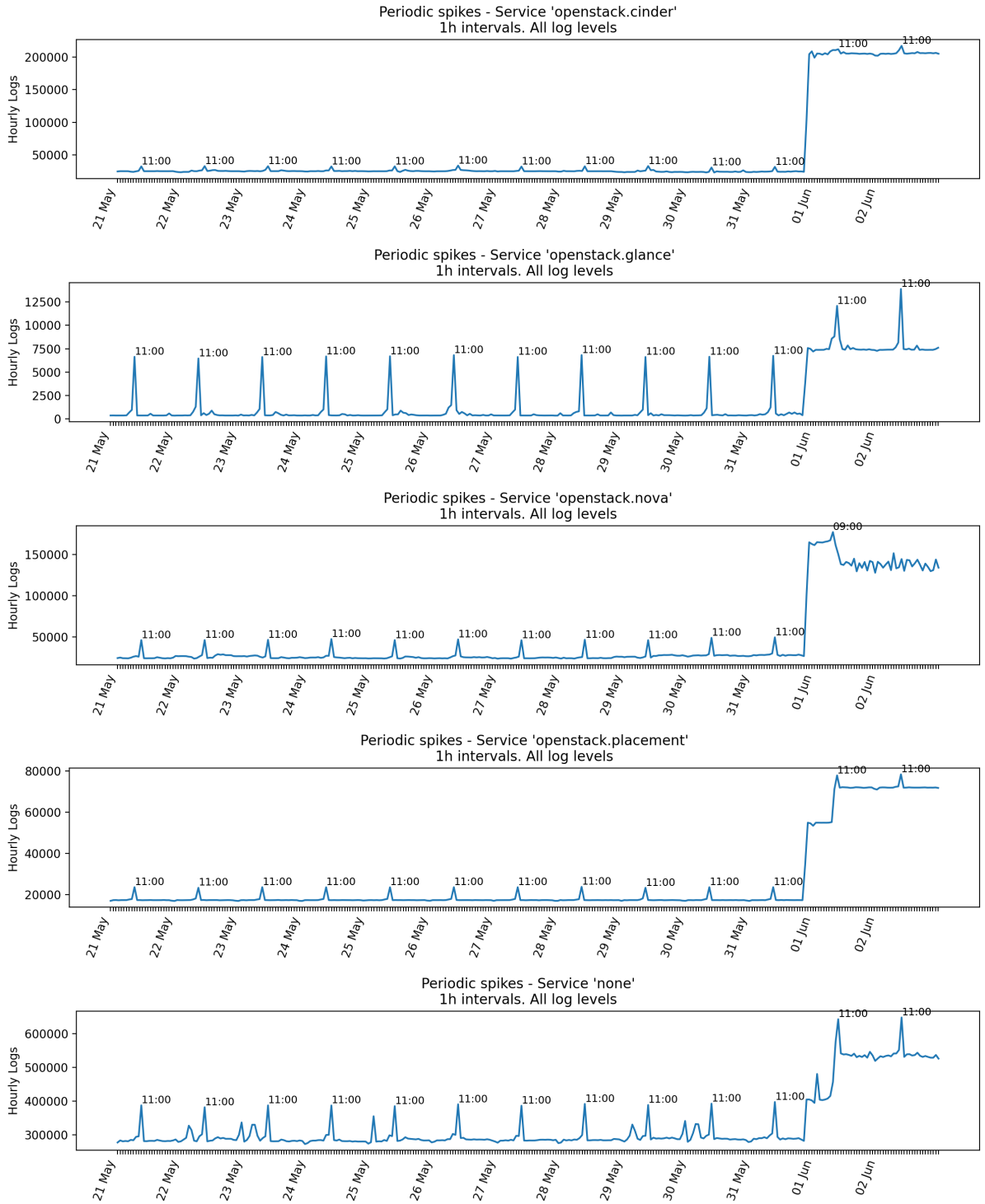


Figure 6.8: Counting logs in hourly intervals to reveal patterns within each service. All of these services show a significant spike at the end of 31 May until the last day. Each service also reveal daily spikes between 11:00 and 12:00. Also, Placement and “none” show another spike during the morning of 1 June.

The next tests used Isolation Forest to analyze log count matrices. Figure 6.9 shows anomaly scores on log count matrices counting service logs in three hour intervals, using all log levels and a threshold of 0.05.

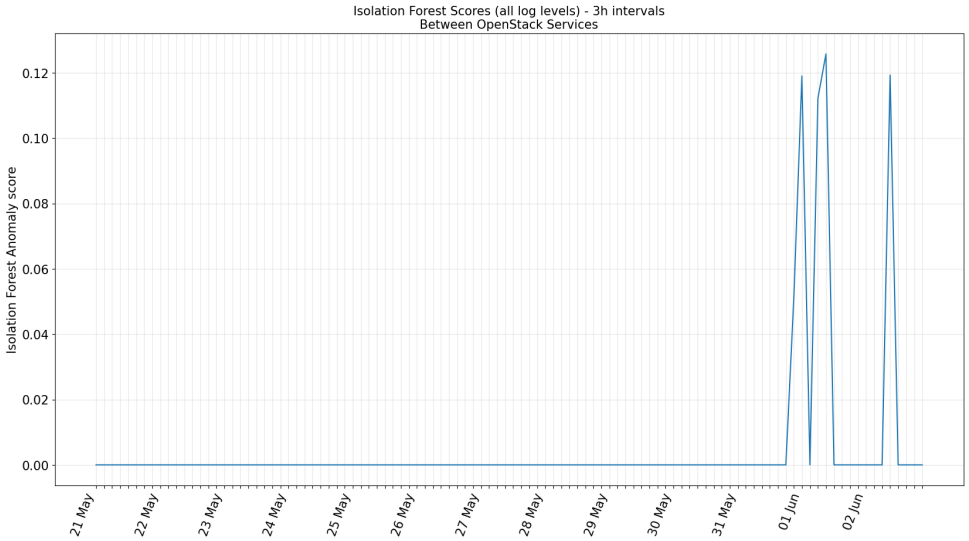


Figure 6.9: Isolation Forest anomaly scores on log count matrix with columns as services. Using all log levels in 3 hour intervals with the threshold 0.05.

To ensure that the more severe log levels are analyzed, each is examined individually, with the logs associated with the *ERROR* log level showing the most significant increase. Figure 6.10 show hourly error logs across all days, where most of them appear between the end of 31 May and the following day.

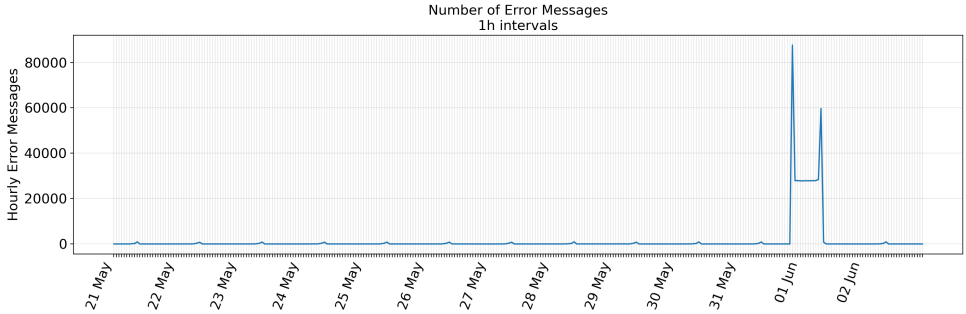


Figure 6.10: Hourly error messages across all days show that nearly all errors occur between the end of 31 May and midday on 1 June.

Continuing on exploring logs associated with the *ERROR* log level, log count matrices are constructed using the categories programs and python modules. Isolation Forest was applied on each log count matrix to produce anomaly scores on each time interval. Figures 6.11 and 6.12 show anomaly scores in three hour log count matrices. Once again, there are large anomaly scores during the last days, with multiple services affected.

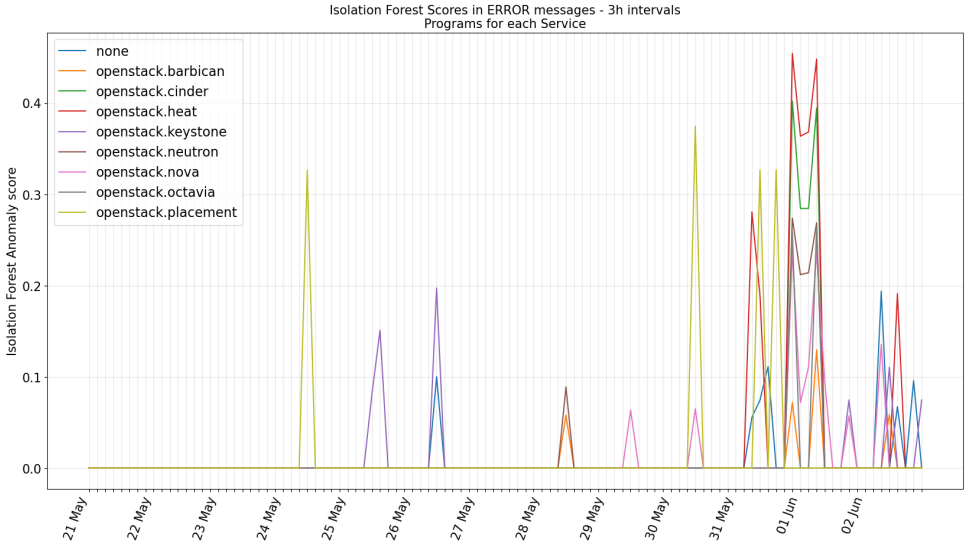


Figure 6.11: Isolation Forest anomaly scores in error logs, with threshold 0.05. Three hour windows counting programnames for each service. Multiple services have high anomaly scores 31 May and onwards.

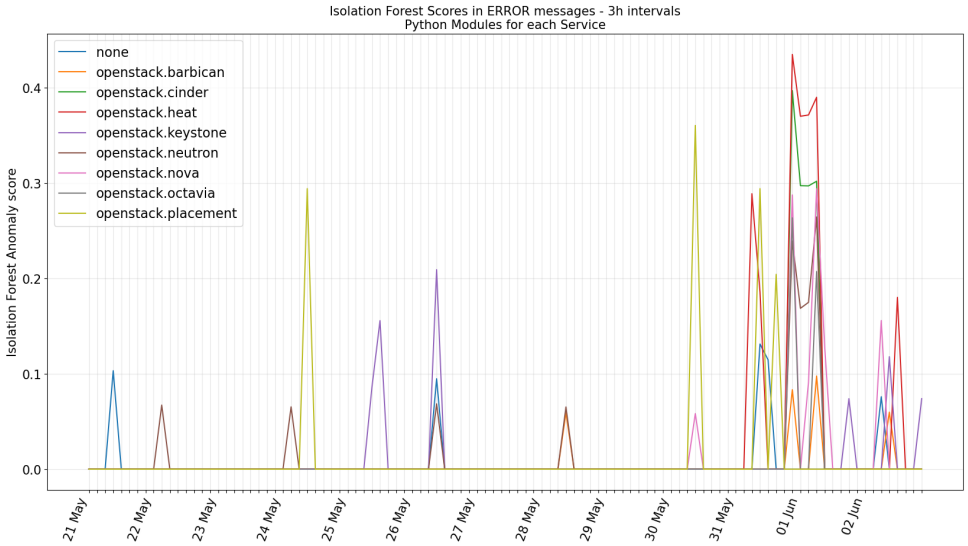


Figure 6.12: Isolation Forest anomaly scores of error logs, with threshold 0.05. Three hour windows counting python modules for each service. Multiple services have high anomaly scores 31 May and onwards.

During the testing, log count matrices are also visualized with heatmaps and manually explored, which sometimes reveals interesting patterns. One of those log count matrices is shown in Figure 6.13, displaying error daily logs across the services. Starting at 31 May until 1 June, there is a significant spike in the services Neutron, Nova, Heat, Cinder, and Octavia.

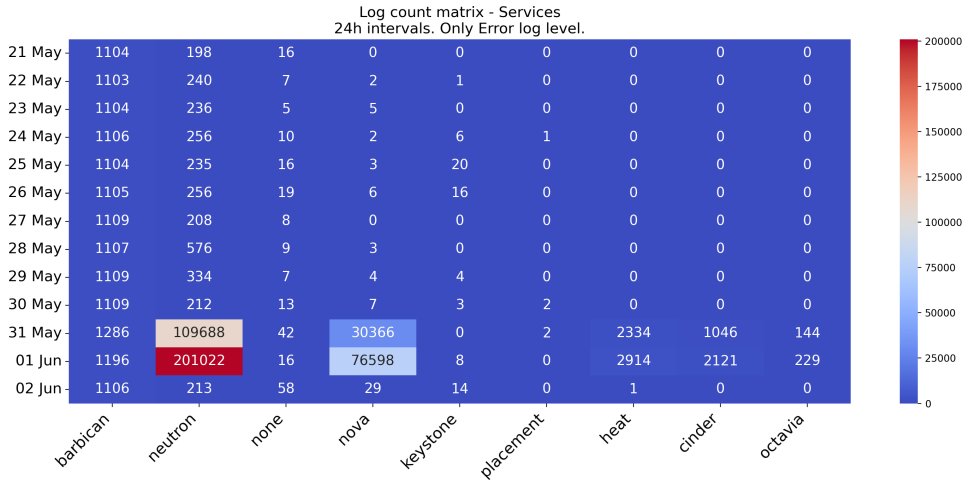


Figure 6.13: A more detailed view of error message distribution across OpenStack services, showing a spike between 31 May and 1 June in the services Neutron, Nova, Cinder, Heat, and Octavia.

Narrowing down on the days 31 May and 1 June using shorter time intervals reveals more detailed information about when the spiking start and ends, shown in Figures 6.14, 6.15, and 6.16.

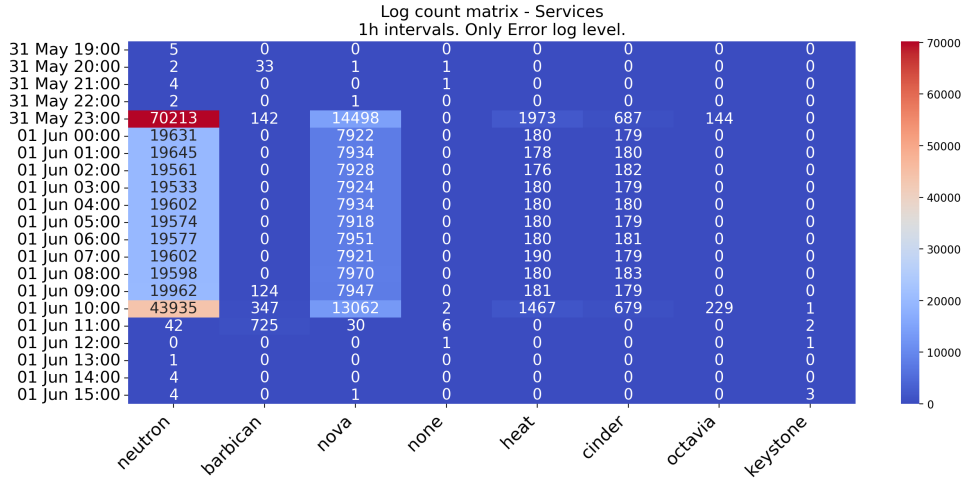


Figure 6.14: Hourly error messages across services. Narrowing down when error spiking starts and ends.

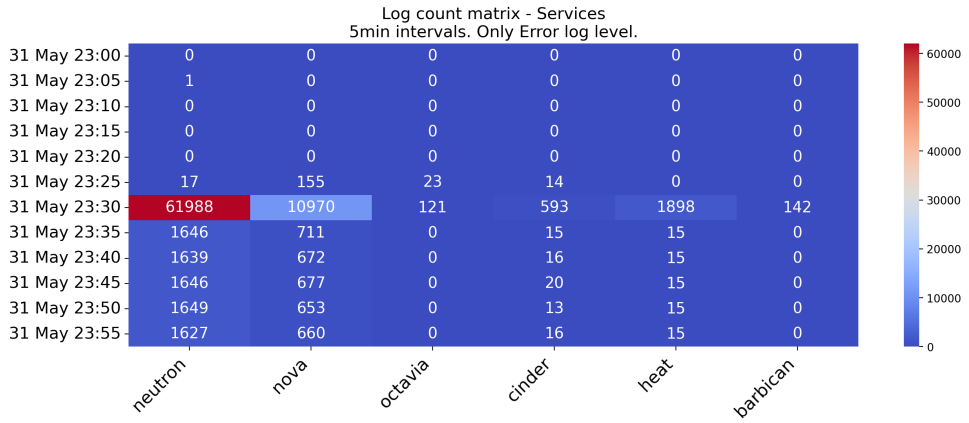


Figure 6.15: Detailed view showing that error spiking starts approximately 31 May 23:30

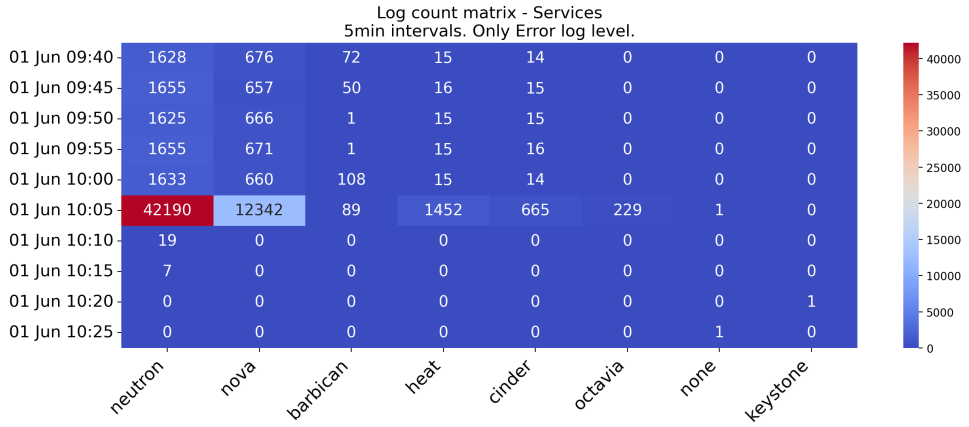


Figure 6.16: Detailed view showing that error spiking ends approximately 1 June 10:05

Closing in on the anomalous time intervals already identified, the next analysis focuses on users. The chosen time interval is the afternoon of 31 May until the afternoon of 1 June to capture several hours before and several hours after the previously identified anomalous interval. The log count matrix uses all log levels and hourly time intervals to get an overview, shown in Figure 6.17

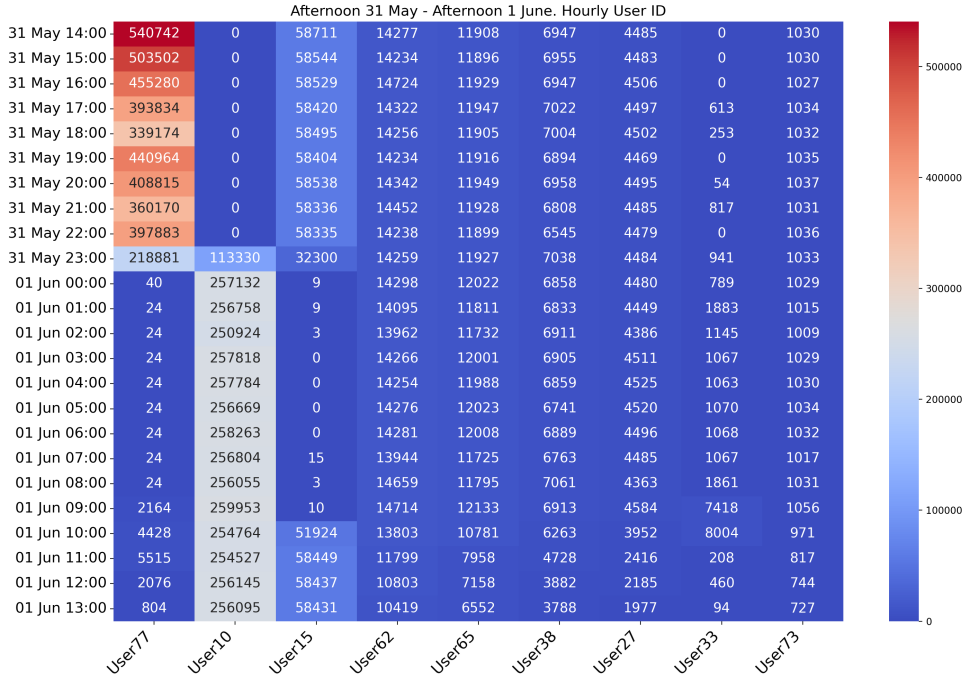


Figure 6.17: Log count matrix of users, capturing several hours before and after the previously identified anomalous time interval. Two of the most voluminous users across all days almost stop to produce logs, while a new user starts generating a large number of logs.

Each anomalous user is analyzed further, and one user is of particular interest, as logs associated with the user appear at the same time as other anomalies in the system, indicating a potential correlation. Figure 6.18 shows services affected by the user in six hour intervals, which shows a large number of messages across multiple services.

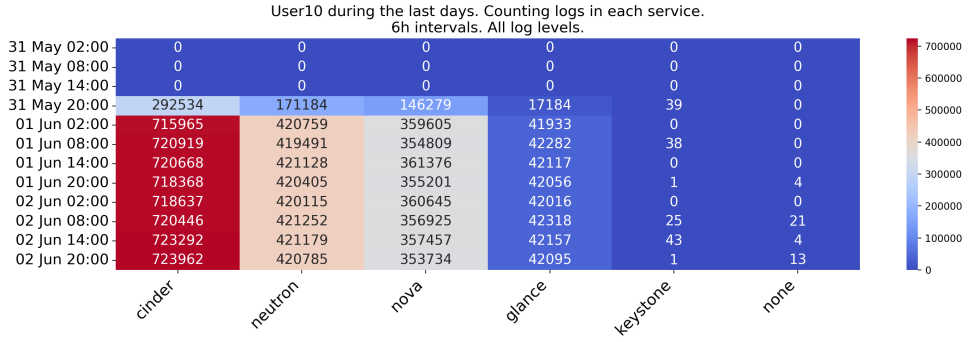


Figure 6.18: One particular user is associated with a massive spike at the same time as other ususual events in the system. Logs associated with the user affect serveral services.

7 Discussion

This section evaluates the results, highlights key findings, and discusses the limitations of this study.

7.1 HDFS Evaluation

Fine-tuning the AE on the HDFS dataset using varying parameters showed that anomaly detection performance improves with bidirectionality, with consistent results across all tested models (see Table 6.2). Bidirectionality mostly affects *Recall*, which is the ratio of correctly predicted anomalies over all anomalies. Using bidirectional networks has been successful in previous research and is once again confirmed. The reason may be that the model learns patterns from both directions of the input data.

Using one-layered LSTMs instead of two-layered also seem to improve anomaly detection, and there is no clear explanation for this. Since only one labeled dataset is included in this study and the fact that only a limited number of parameters are used during fine-tuning, it is difficult to explain why one-layered LSTM perform slightly better.

For the best performing models using bidirectionality and single-layered networks, the differences in F1-score are negligible. However, the model with 192 intermediate compression size shows the highest F1-score, due to a slightly higher precision score.

7.2 ERDC Evaluation

The initial analysis was conducted to examine the payload lengths in the log files, to ensure that nothing unusual was present in the log files. This analysis revealed that the last three days had larger average payload lengths than the other days, and upon further investigation it was found that those messages were directly related to system issues, and those messages were longer than the average messages across all days. The analysis also explored the existence of potential multibyte characters in the log files, and this showed that they do exist, but only in messages that mask user information, and the size of the extra bytes is the same each day. Going all the way back to the raw log files did not reveal anything already found during the analysis of the payload lengths, and the reason is possibly due to an even spread of most categories, meaning that existing and non-existing values are distributed evenly across all days.

For most of the time intervals, there are no significant differences in the number of messages generated by the system, possibly because the services continuously generate log messages with high velocity even when users are idle.

Counting logs in time intervals over all services and between services revealed interesting patterns in many services. Hourly windows were sufficient to capture patterns without too much noise, and it could be confirmed that a spike happened at the same time for several services. Also, at the same time a drop in number of messages occurs in the most voluminous service *Neutron*, whichs shows a significant lower

number of log messages until a significant spike occurs the next morning, where suddenly the number of messages increases significantly. The seemingly unaffected services are *Barbican*, *Horizon*, and *Heat*.

Selecting logs with higher log severity level than *WARNING* revealed a spike in number of messages associated with the *ERROR* log level during the evening of the third-to-last day, and upon closer investigation, a connection was lost to a distributed message queue service called *RabbitMQ*, causing an outburst of similar messages generated from multiple services dependent on the distributed message queue, and related messages were continuously generated during the following night, displaying that each attempt to connect to the queue was timed out.

To conclude the findings of the ERDC results, at the end of the third-to-last day one user is associated with a large number of requests that span across the services Cinder, Neutron, Nova, and Glance, possibly due to the use of an automation tool. Simultaneously, a distributed messaging queue becomes unavailable for system components as well as diagnostic tools running in the system, such as prometheus components.

It should be noted that in the Chapter 6.5, there are no results from the sequential analysis of the ERDC dataset. The reason for this is mainly because of the complex nature of the logs requiring sophisticated regular expressions, making the log parsing inaccurate and invalidating such an analysis. Therefore, the testing on the ERDC dataset is performed without log IDs and log templates.

7.3 Key Findings

Providing only a few regular expressions to the Drain log parser was not sufficient when parsing a large-scale dataset with high entropy.

Using SBERT embeddings without fine-tuning achieved results comparable to other research.

Using bidirectional LSTMs improved anomaly detection performance.

Intermediate compression size did not significantly improve the anomaly detection performance.

Using a threshold set to the average loss plus two standard deviations was optimal across all tested threshold values.

7.4 Limitations

This study has shown that the designed AE has achieved promising anomaly detection performance on the publicly available HDFS dataset, but there is no guarantee that this holds for other datasets, and even if the publicly available datasets are labeled by domain experts, the datasets have limitations, as described by Landauer et al. [40]. Furthermore, the study uses a single dataset, and it would have been more comprehensive to include additional labeled datasets. The main reason for using a single dataset is time constraints, as the study has two main goals: developing and testing an anomaly detection model and exploring a large-scale dataset.

The large-scale ERDC dataset has high entropy, resulting in low parsing accuracy when using a log parser. Since rerunning the parser on the dataset takes approximately two weeks, the approach was to manually sift through the existing log templates and merge those that seemed similar, which slightly improved the accuracy, but not enough to be used in a sequential analysis dependent on accurate templates.

Log anomaly detection involves many steps, each with multiple parameters to tune, leading to parameter sprawl, and unfortunately the time has been too limited to test multiple values for each parameter.

8 Conclusion

This study focuses on unsupervised log anomaly detection, which involves several subtasks before any results can be obtained, subtasks such as log parsing, feature extraction, unsupervised training, and anomaly classification.

Based on the findings of the literature study, an anomaly detection method is designed. The method uses the Drain log parser to classify each log message into a log group, and each log group has a unique log ID and a log template. For feature extraction, the LLM SBERT converts each log template into a numerical embedding vector. In this way, sequences of log templates become embedding matrices, which can be used in downstream tasks. The choice was then between designing a prediction-based or a reconstruction-based method, and there exists research suggesting that reconstruction-based methods are more robust when training data contain anomalies. The reconstruction-based method uses an LSTM autoencoder (LSTM-AE) as the deep learning model to train on embedding matrices. LSTMs are chosen because they handle longer log sequences without losing information. The AE is trained on embedding matrices to learn patterns in the data and reconstruct them. During classification, new matrices are passed through the trained AE, and those matrices with high reconstruction errors are more likely to contain anomalous events or sequences.

The designed method is benchmarked on a labeled HDFS dataset, chosen because of its popularity in research, making it easier to compare the results with other research. The results show that the proposed method performs comparably with other research, especially when the AE uses bidirectional LSTMs.

In a collaboration with an external company, the study also includes an analysis of their dataset, which is a real large-scale OpenStack dataset comprising log files of 13 consecutive days and in total over 500 million log messages. The company noticed larger log files in the last days and asked me to analyze the data to determine the cause of the growth. The dataset is not only large but has high entropy, which makes log parsing challenging. Since the Drain parser depends on provided regular expressions, parsing such a dataset with many regular expression patterns is computationally intensive. As a result, the analysis has been done without relying on log IDs or log templates from the parser. Instead, the work has focused on coarse-grained analysis to narrow down where to examine the logs more closely.

8.1 Future Work

Parsing large-scale high-entropy logs is challenging, and a study could be to choose such a dataset and design a log parser and compare it with a few state-of-the-art parsers, and evaluate the performance both in terms of accuracy and efficiency. As described in this study, sequential parsing of large-scale logs is time-consuming, and a possibility could be to design a parser with scalable parallel computing.

For prediction-based anomaly detection methods that use transformer-based BERT, the use of distilled BERT models is an interesting approach for future research. There are many distilled models to explore in log anomaly detection, such as DistilBERT, TinyBERT and MobileBERT, and a suitable direction could be to incorporate them in a similar manner as popular methods using BERT, and compare them to methods

using BERT in terms of accuracy and efficiency. This was explored in the study but was later discontinued, as the focus changed to designing a reconstruction-based method instead.

The next one is quite interesting but requires a well-designed evaluation; use LLMs to summarize large sequences of log messages to describe what is happening. The motivation for this is twofold: (1) prevent engineers from reading large amounts of log messages during monitoring. (2) when matching messages with a parser, rare log messages may not be recognized and there could be thousands or millions of unclassified log messages, and a summary of these text chunks could be beneficial in decision-making. This was explored in this study, but was discarded due to the difficulty of assessing the performance.

Another interesting idea is to choose existing state-of-the-art anomaly detection methods — both prediction-based and reconstruction-based — and apply them on publicly available datasets. These datasets could then be polluted with varying amounts of anomalies to create more realistic scenarios, and then train the methods on anomalous data and evaluate which methods perform best at detecting anomalies. While such studies already exist, they become outdated as new methods continuously emerge. Incorporating anomalies into the training data and compare the performance with other methods was considered in this study, but discarded, since many state-of-the-art methods are either only abstractly described and require careful parameter tuning, or they are not publicly available from the original authors and instead implemented by third-party developers, making the process overly complex. As a result, designing such a study would require careful planning to decide which methods to include and how to evaluate them properly to make them justice.

Another suggestion is to explore ways to efficiently handle OOV words when using LLMs in log anomaly detection, since this is one of the weaknesses when the models are utilized in domain-specific language such as log messages. One such idea could be to design a large corpus of logs from multiple datasets and train an LLM on the corpus before extracting embeddings, and use an existing state-of-the-art anomaly detection method and assess if the trained embeddings improves anomaly detection performance.

8.2 Personal Reflection

Log analysis is a challenging topic and crucial for service providers to improve their services. I have never worked with log analysis and I am glad that I was given the opportunity to learn more about it. The tools used in this study seem popular in the industry, such as the Python programming language, the Python library Pandas, the infrastructure/interface JupyterHub/JupyterLab, and to further explore ML and LLMs. The gained experience might be useful in the future.

The direction of the project changed multiple times during the study. Initially, to use LLMs to summarize a sequence of logs, which would be helpful to engineers, but the issue was the lack of validation sets to assess the accuracy of the results and would require engineers to evaluate each summary. Another direction was to design a prediction-based method using smaller LLMs, such as DistilBERT, since BERT is widely used in anomaly detection, and it would be interesting to compare DistilBERT with BERT, but the proposed model is a reconstruction-based method which incorporates LLMs for the sole purpose of extracting embeddings (and possibly for brief fine-tuning on templates). Therefore, the bottleneck is not the use of LLMs, and incorporating DistilBERT for efficiency makes no sense. The next direction was to evaluate the quality of the embeddings for each log template, but the problem was the lack of validation tools. There are existing tools for well-formed English, but not for the domain-specific language used in log messages.

Working with a real OpenStack dataset has been a valuable experience, and compared to the small and

structured datasets available online, there are many differences worth mentioning. The high entropy is obnoxious as the log formats are so varied and unpredictable, and they contain information not present in publicly labeled datasets. Timestamps are all over the place and in so many different formats, requiring complex regular expressions. There are also shell messages written with ansi escape sequences, supposedly to format terminal outputs, but when used in log parsing get messed up as gibberish in the log templates, and there are also quite odd test messages logged each day. There are also extremely large messages that print entire python lists, python sets, python dictionaries, and entire parameter lists of python functions called. It is not easy for a student like me to understand how to write these regular expressions to avoid matching too much or too little, and the lists of regular expressions can grow quickly, making the regular expression matching slow. I have written regular expressions for the unmatched OpenStack logs in an attempt to provide these matches as input to Drain to merge into the previously matched OpenStack logs, and this work has been difficult and time-consuming.

It has been an enriching experience to work with both log parsing and log anomaly detection, but in hindsight, I should have narrowed down the scope to focus solely on log parsing, mainly due to the issues with log parsing accuracy on the large-scale ERDC dataset. This study has shown that parsing a voluminous log dataset with high entropy is challenging and should have been the endpoint of this study.

References

[1] Opensource.com. What is openstack? https://opensource.com/resources/what-is-openstack, 2025. [Accessed 06-04-2025].

[2] Wikipedia.org. OpenStack - Wikipedia. https://en.wikipedia.org/wiki/OpenStack, 2025. [Accessed 06-04-2025].

[3] Freecodecamp.org. OpenStack Tutorial — Operate Your Own Private Cloud. https://www.freecodecamp.org/news/openstack-tutorial-operate-your-own-private-cloud/, 2025. [Accessed 06-04-2025].

[4] Openstack.org. OpenStack Releases: 2023.1 Antelope — releases.openstack.org. https://releases.openstack.org/antelope/, 2025. [Accessed 06-04-2025].

[5] OpenStack. Open Source Cloud Computing Platform Software - OpenStack — openstack.org. https://www.openstack.org/software/, 2025. [Accessed 31-03-2025].

[6] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News*, 44(2):489–502, 2016.

[7] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Run-time failure detection via non-intrusive event analysis in a large-scale cloud computing platform. *Journal of Systems and Software*, 198:111611, 2023.

[8] OpenStack Documentation. Design 2014; arch-design 0.0.1.dev15 documentation — docs.openstack.org. https://docs.openstack.org/arch-design/design.html, 2025. [Accessed 12-05-2025].

[9] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and Benchmarks for Automated Log Parsing. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 00:121–130, 2019. doi: 10.1109/icse-seip.2019.00021.

[10] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021. ISSN 0360-0300. doi: 10.1145/3460345.

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An Online Log Parsing Approach with Fixed Depth Tree. *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017. doi: 10.1109/icws.2017.13.

[12] Yukyung Lee, Jina Kim, and Pilsung Kang. LAnoBERT: System Log Anomaly Detection based on BERT Masked Language Model. *arXiv*, 2021. doi: 10.48550/arxiv.2111.09564.

[13] Junchen Ma, Yang Liu, Hongjie Wan, and Guozi Sun. Automatic Parsing and Utilization of System Log Features in Log Analysis: A Survey. *Applied Sciences*, 13(8):4930, 2023. doi: 10.3390/app13084930.

[14] Song Chen and Hai Liao. BERT-Log: Anomaly Detection for System Logs Based on Pre-trained Language Model. *Applied Artificial Intelligence*, 36(1):2145642, 2022. ISSN 0883-9514. doi: 10.1080/08839514.2022.2145642.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882.

[17] Peter J Rousseeuw and Mia Hubert. Anomaly detection by robust statistics. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1236, 2018.

[18] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[19] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[20] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth ieee international conference on data mining*, pages 413–422. IEEE, 2008.

[21] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):1–39, 2012.

[22] Wiki. Long short-term memory - Wikipedia — en.wikipedia.org. `https://en.wikipedia.org/wiki/Long_short-term_memory`, 2025. [Accessed 05-05-2025].

[23] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[24] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 151–158. IEEE, 2018.

[25] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[26] Wiki. Gated recurrent unit - Wikipedia — en.wikipedia.org. `https://en.wikipedia.org/wiki/Gated_recurrent_unit`, 2025. [Accessed 05-05-2025].

[27] Haixuan Guo, Shuhan Yuan, and Xintao Wu. LogBERT: Log Anomaly Detection via BERT. *2021 International Joint Conference on Neural Networks (IJCNN)*, 00:1–8, 2021. doi: 10.1109/ijcnn52387.2021.9534113.

[28] SBERT.net. Pretrained Models; Sentence Transformers documentation — sbert.net. https://www.sbert.net/docs/sentence_transformer/pretrained_models.html, 2025. [Accessed 04-05-2025].

[29] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R Lyu. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. *arXiv*, 2021. doi: 10.48550/arxiv.2107.05908.

[30] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[31] Aarish Grover. Anomaly detection for application log data. *ScholarWorks*, 2018.

[32] Sangkeum Lee, Hojun Jin, Sarvar Hussain Nengroo, Yoonmee Doh, Chungho Lee, Taewook Heo, and Dongsoo Har. Smart metering system capable of anomaly detection by bi-directional lstm autoencoder. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6. IEEE, 2022.

[33] Hasan Torabi, Seyedeh Leili Mirtaheri, and Sergio Greco. Practical autoencoder based anomaly detection by using vector reconstruction error. *Cybersecurity*, 6(1):1, 2023.

[34] Ahmed Shoyeb Raihan and Imtiaz Ahmed. A bi-lstm autoencoder framework for anomaly detection-a case study of a wind power dataset. In *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, pages 1–6. IEEE, 2023.

[35] Caiping Hu, Xuekui Sun, Hua Dai, Hangchuan Zhang, and Haiqiang Liu. Research on log anomaly detection based on sentence-bert. *Electronics*, 12(17):3580, 2023.

[36] LogPai. loghub/HDFS at master · logpai/loghub — github.com. https://github.com/logpai/loghub/tree/master/HDFS, 2025. [Accessed 06-04-2025].

[37] Apache. HDFS Architecture Guide — hadoop.apache.org. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2025. [Accessed 06-05-2025].

[38] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 355–366. IEEE, 2023.

[39] LogPai. loghub/HDFS/HDFS_templates.csv at master · logpai/loghub — github.com. https://github.com/logpai/loghub/blob/master/HDFS/HDFS_templates.csv, 2025. [Accessed 26-05-2025].

[40] Max Landauer, Florian Skopik, and Markus Wurzenberger. A critical review of common log data sets used for evaluation of sequence-based anomaly detection techniques. *Proceedings of the ACM on Software Engineering*, 1(FSE):1354–1375, 2024.

Appendix

A AE Fine-Tuning

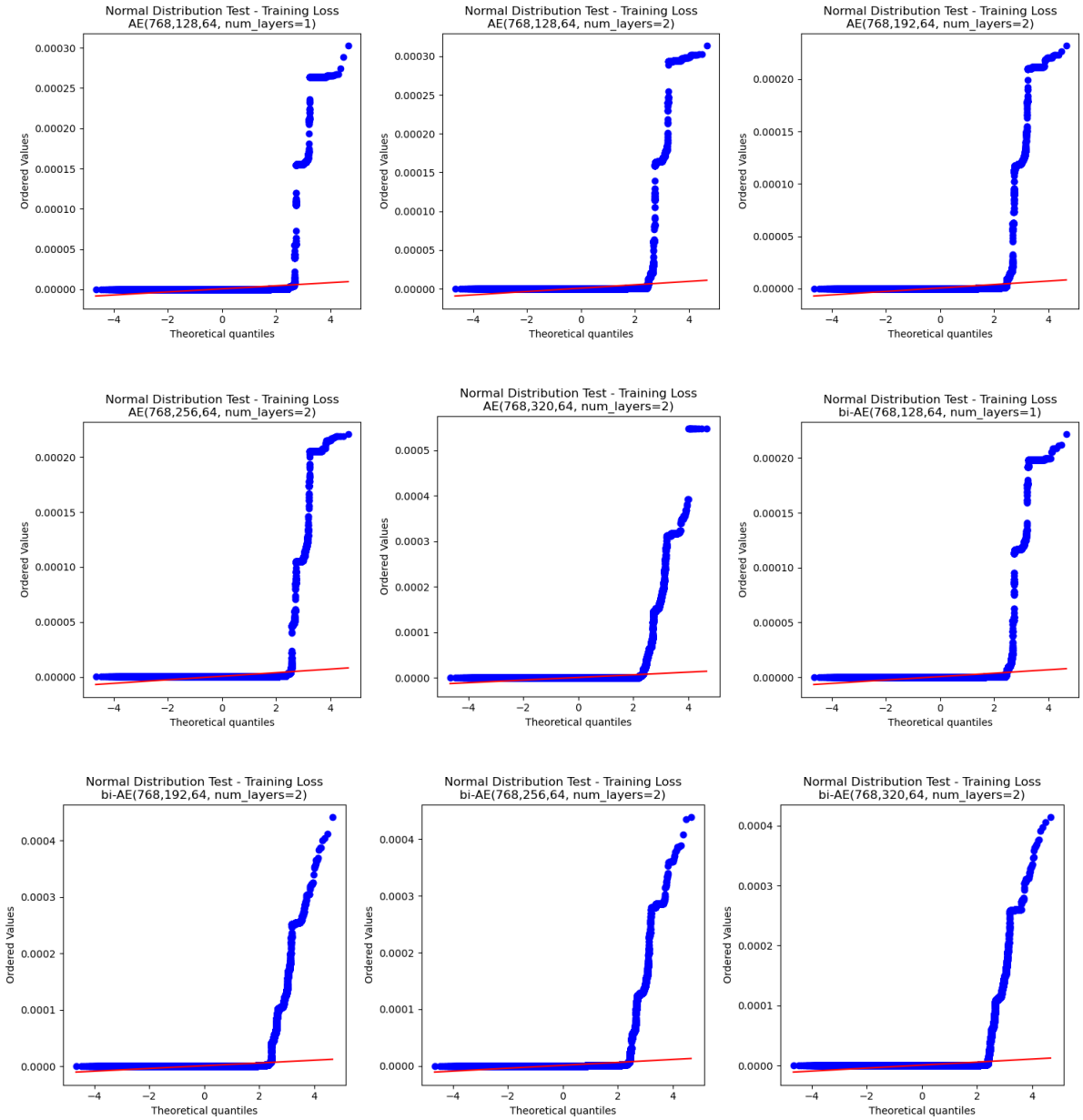


Figure A.1: Normal distribution tests of training loss values for a set of models. The blue dots should increase linearly close to the red line, which none of them do. Therefore, the loss values are not normally distributed.

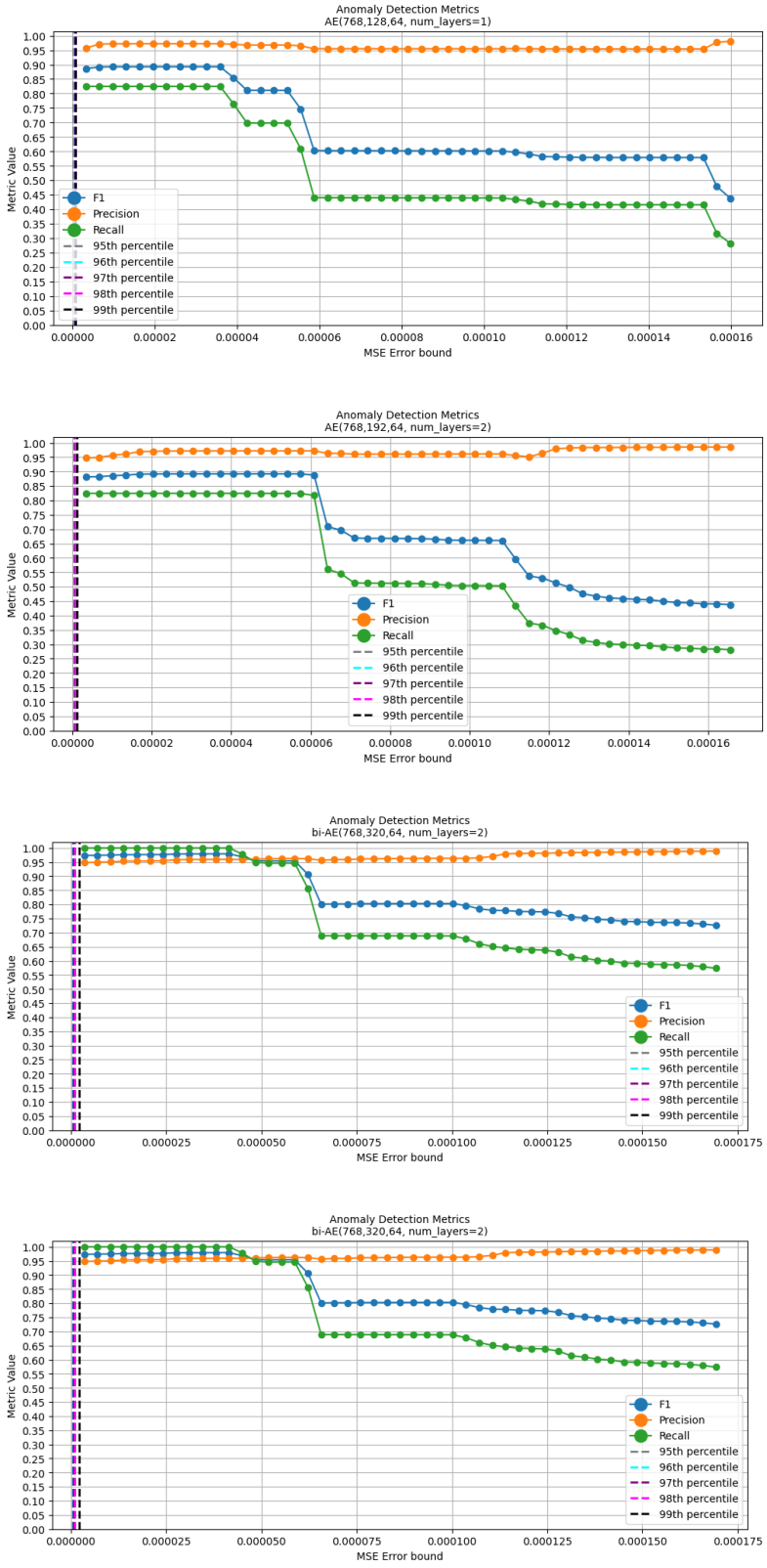


Figure A.2: A set of models during fine-tuning reconstruction thresholds using upper percentiles of training loss values, but these values are too small.

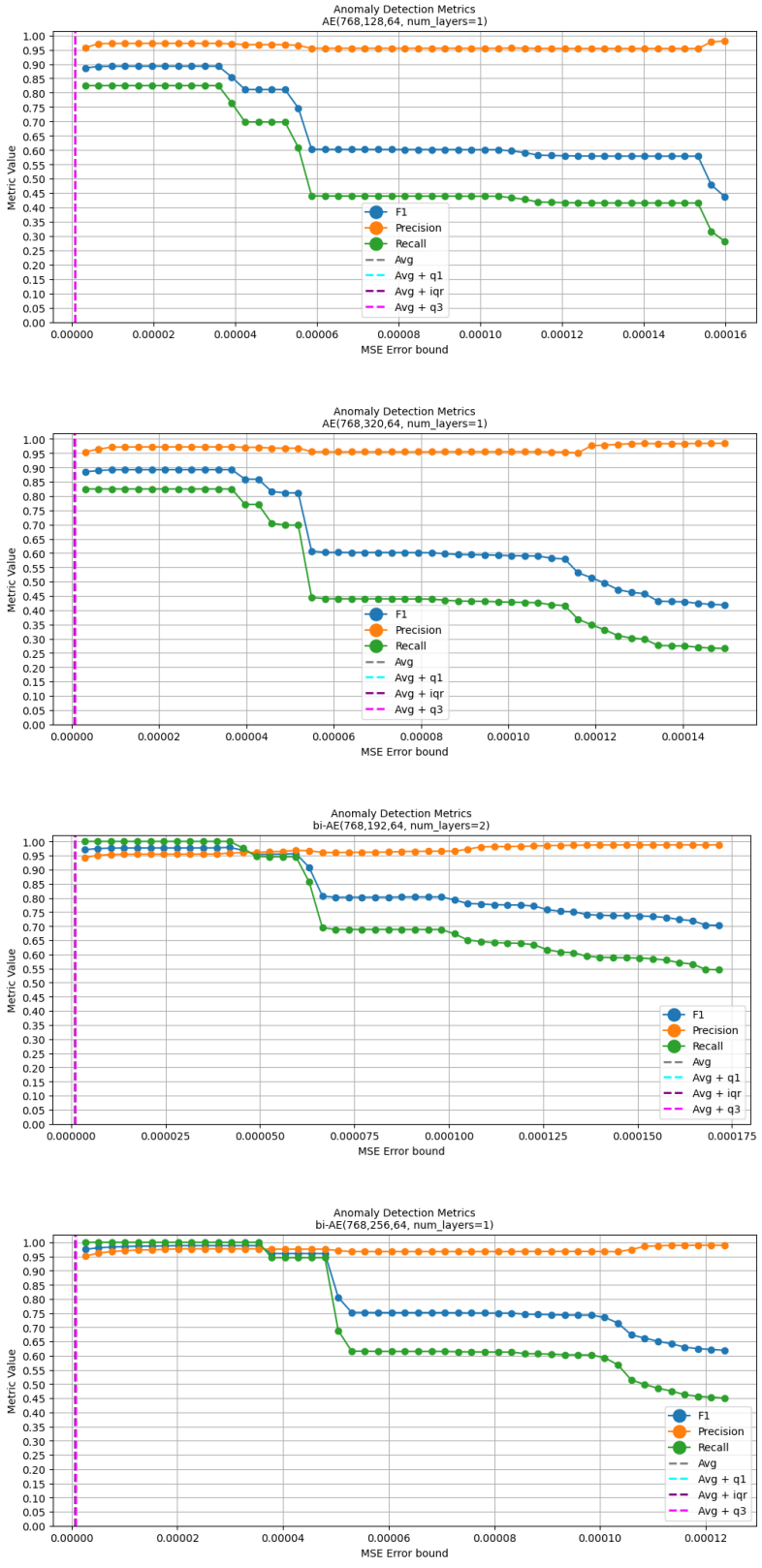


Figure A.3: A set of models during fine-tuning thresholds based on training loss values, using average loss plus quartiles, which are too small thresholds.

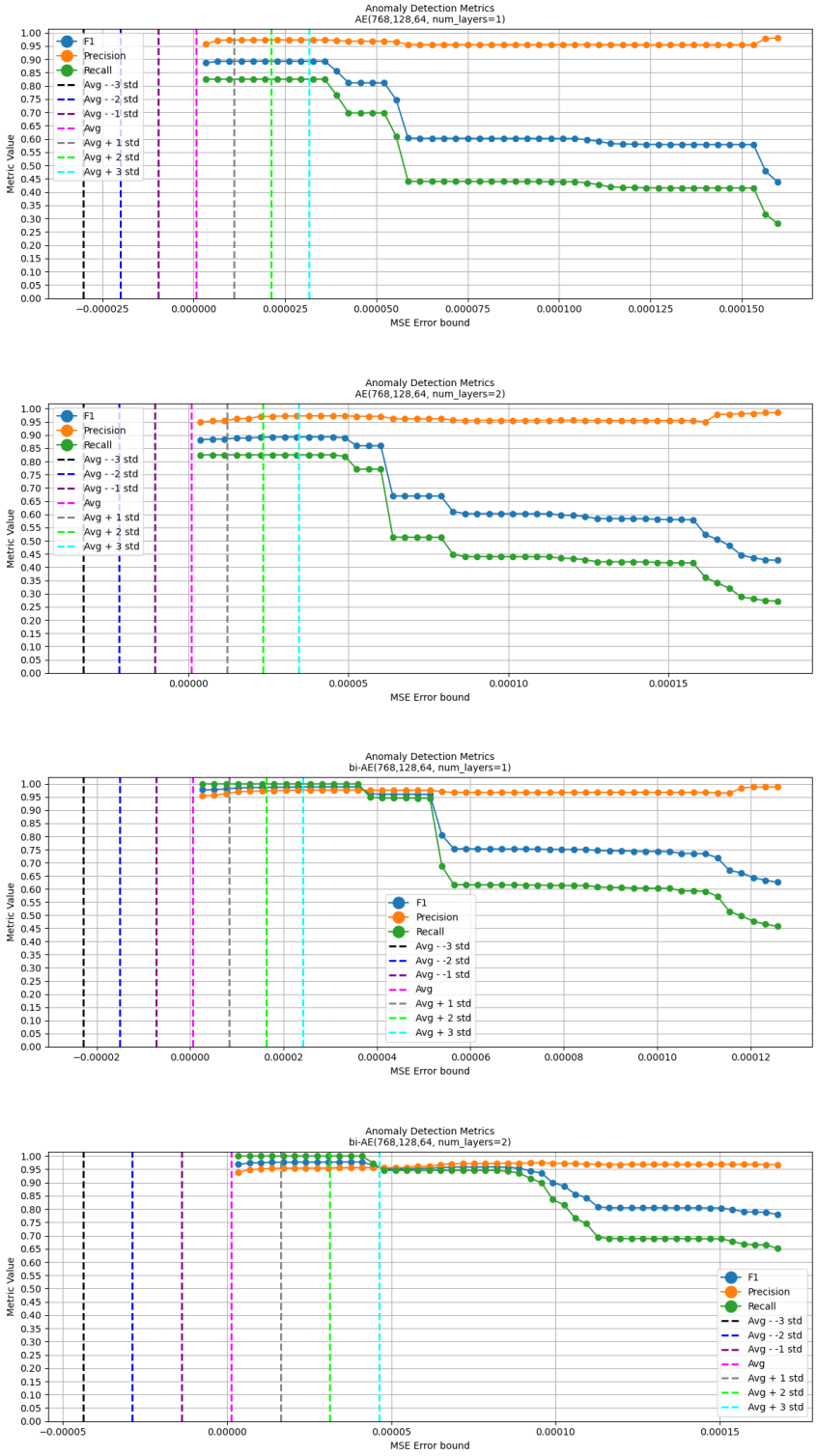


Figure A.4: A set of models during fine-tuning thresholds based on training loss values, using average loss plus standard deviations, which fit well into establishing a threshold. Average loss plus two standard deviations is optimal across all tested models. It should be noted that using standard deviations usually require normally distributed data.

B ERDC Regular Expressions

```
r"(?<=Device Port)\((.+)\)"
r"\[[^\[\]]*\]" #python list
r"(?<==)\[[^\[\]]*\]" #python list w preceding equal sign
r"(?<!:)\{[^:{}]+\}" #python set
r"\{(?:[^{}]|\{(?:[^{}]|\{(?:[^{}]|\{[^{}]*\})*\})*\})*\}" #python dict
r"(?<=Trigger reload_allocations for port )[\s\S]*"
r'(?<="(?:GET|POST|PUT|DELETE)\s)[^"]+' #request strings incl. http ver
r'(?<=(?:GET|POST|PUT|DELETE)\s)[^\n\r]*'
r"(?<=Traceback \(most recent call last\):)[\s\S]*"
r'"[^"]*(?:Mozilla|AppleWebKit|Chrome|Safari|Firefox|Edge|Opera)[^"]*"' #browser user-agent
r"\b(?:https?|ftp|file|ssh|sftp)://\S+"  #url initial pattern
r"(?:/|\.{1,2}/)(?:[\w.-]+/)*[\w.-]+" #file/directory paths
r'(?<=\b(?:user|project|domain):\s)[a-zA-Z0-9._-]+' #users / projects
r"\b\d+(?:\.\d+){1,2}(?:[-._]?(?:dev|post|rc|b|a|alpha|beta|build)?\d*)?(?!\\)\b"
r"\beselda\d{2}u[\w.-]*\b" #hostnames
r"\b(?:\d{1,3}\.){3}\d{1,3}(?:,(?:\d{1,3}\.){3}\d{1,3})*\b" #multi or single IP
r"\b[a-zA-Z0-9_]+-[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\b" #UUID
r"\b[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\b" #UUID without prefix
r'\b[0-9a-fA-F]{32}\b' #compact UUID
r'"HEADVT[\s\S]*?(?:"|$)'
r'(?<=\becho\s*)"[^"]*"'

# all kinds of timestamp and timezones
r"""(?ix)(
(?:\b(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+\d{1,2}\s+\d{2}:\d{2}:\d{2}\b) |
(?:\d{2}/[A-Za-z]{3}/\d{4}:\d{2}:\d{2}:\d{2}(?:\ [+-]\d{4})?) |
(?:\d{4}-\d{2}-\d{2}[T ]\d{2}:\d{2}:\d{2}(?:\.\d+)?(?:Z|[+-]\d{2}:?\d{2})?) |
(?:\d{4}[-/]\d{2}[-/]\d{2}) |
(?:\d{2}[-/]\d{2}[-/]\d{4}) |
(?:\d{8}(?:\d{6})?) |
(?:\b\d{2}:\d{2}(?::\d{2})?\b))"""

r"(?<=Stack CREATE COMPLETE\s+)\((.+)\)"
r'(?<=: Invalid method in request\s)\S+'
r'(?<=est-selfservice-booking)[\w.-]+'
r"(?<=net_)[\w]+"
r"(?i)\b(?:[0-9a-f]{2}([-:]))(?:[0-9a-f]{2}\1){4}[0-9a-f]{2}\b|\b[0-9a-f]{12}\b"
r"\b(?:\d{1,3}\.){3}\d{1,3}:\d{1,5}\b"
r'(?<=\bGET\s)/[^\s]+'
r"[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}"
r"\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}Z"
r"(?i)[0-9a-f]{32}"
r"[+-]?\d+(\.\d+)?([eE][+-]?\d+)?"
r"(?i)([0-9a-f]{2}:){5}[0-9a-f]{2}"
r'\b[\w.]*exceptions?\.[\w.]*\b'
r'\w+(?=_external_(?:front|back))'
r'(nova|neutron|glance|heat|keystone|cinder|swift|octavia|barbican|placement|manila|trove|mistral|ironic)'
r'.*(?=UserWarning: Policy \S+ failed scope check)'
r'(?<=got incomplete line before first line from ).*'
r"(?<=VIFBridge)\((.+)\)"
r"(?<=AFTER events, as in 'AFTER it's committed', not BEFORE.\s+).*"
r"(?<=receive tunnel port not found\s+).*"
r"(?<=privsep process running with capabilities\s+).*"
r'(?<=required)(?=\s+warnings).*'
```

C ERDC Testing

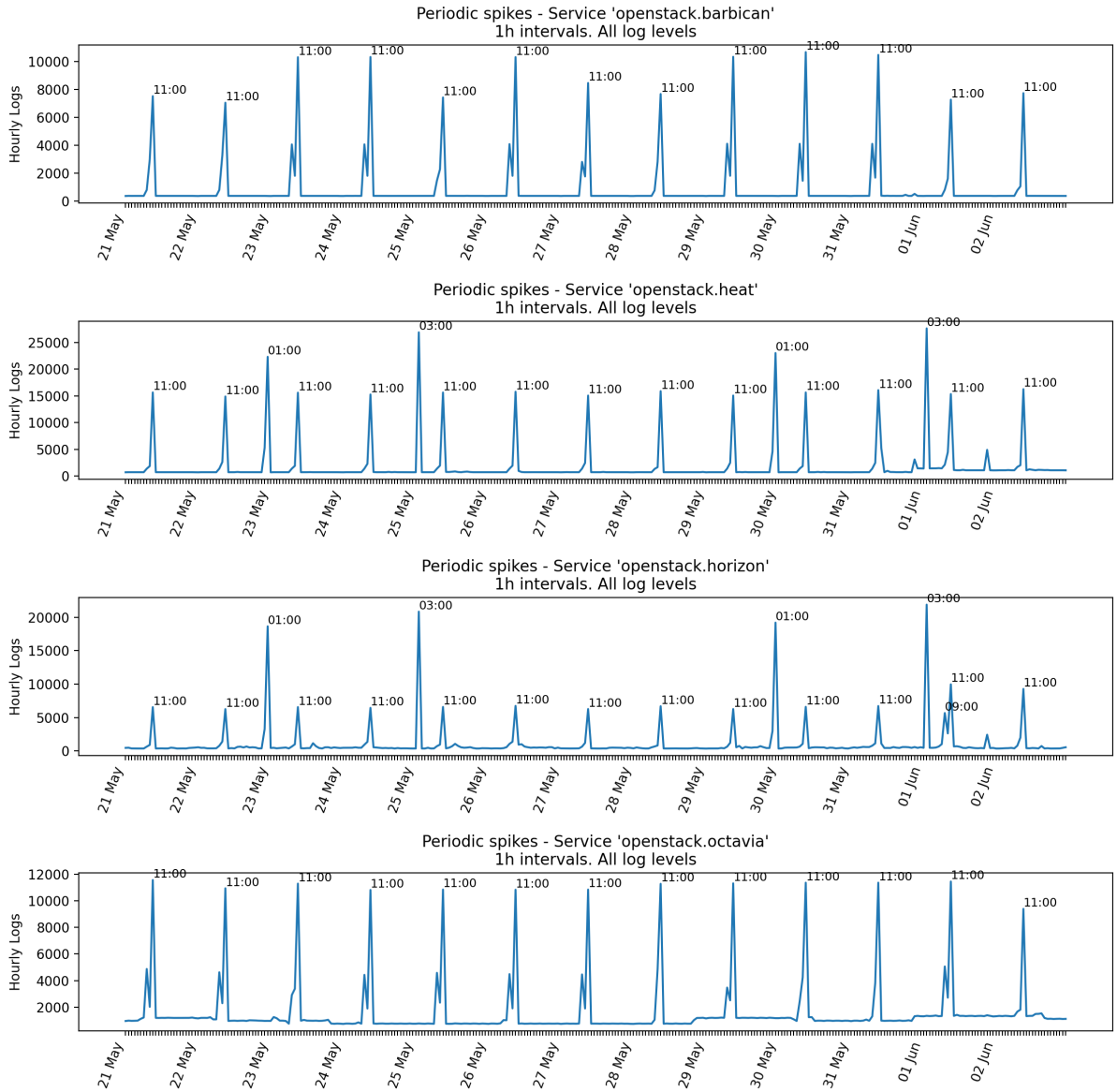


Figure C.1: Counting logs in hourly intervals. Barbican, Heat, Horizon, and Octavia spiking daily 11:00 – 12:00. Heat and Horizon show weekly spikes 01:00 each Tuesday and 03:00 each Thursday. Also, all these four services seem quite unaffected during the last days.

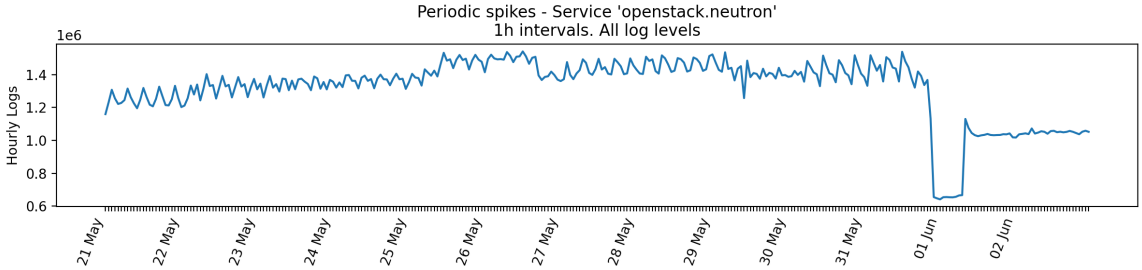


Figure C.2: Neutron shows no daily spike pattern. A sudden drop occurs at the end of 31 May, followed by several hours of lower hourly messages before a spike that does not reach the levels before the drop.

The Keystone service has a slightly increasing number of logs during the last days, but not as significant as the services shown in Figure 6.8.

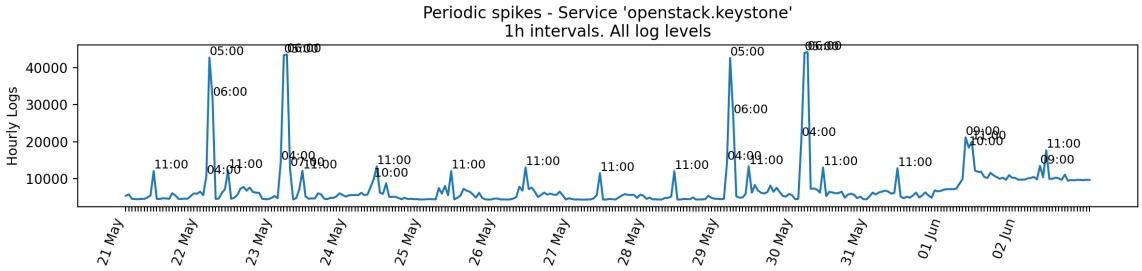


Figure C.3: Counting keystone logs in hourly intervals. The service shows a slightly increase in logs during the last days.