Aalto University
School of Electrical
Engineering

Master's programme in Automation and Electrical Engineering

Log Analysis and Anomaly Detection in Log Files with Natural Language Processing Techniques

Kamal Dhakal

Master's Thesis
2023

Aalto University
School of Electrical Engineering

Author Kamal Dhakal

Title Log Analysis and Anomaly Detection in Log Files with Natural Language Processing Techniques

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems

Supervisor Prof. Paavo Alku

Advisors Dr Kaveh Samiee, Mr Meng Sun (MSc)

Collaborative partner GE Healthcare

Date 9 July 2023 **Number of pages** 78 **Language** English

Abstract

Log analysis is a crucial aspect of maintaining and improving the performance, security, and reliability of modern computer systems. The increasing complexity of these systems along with the exponential growth of log data has driven the need for the development of more advanced techniques for understanding and analyzing logs. In this project, we propose a log management infrastructure with Elastic Stack for statistical analysis equipped with visualization features and natural language processing (NLP) based approaches for the process of log analysis and anomaly detection. We build upon a classification model with 4 different classes on a small sampled dataset to develop a proof-of-concept (POC) to validate that the proposed solution aligns with the problem statement. We then scale up the solution to the full dataset to develop anomaly detection in real-world syslog data generated in industrial settings. This enables faster and more effective decision-making which in turn frees the human workforce from the manual repetitive process of log inspection. First, raw textual and unstructured logs in various formats and from different sources such as Continuous Integration/Continuous Deployment (CI/CD) servers, ambulatory monitoring devices, and automated test builds are collected. Then, the obtained logs are preprocessed to clean, normalize, and tokenize into tokens. The tokenization is carried out using word and sub-word tokenization techniques to obtain word and sub-word tokens respectively. The tokens are then converted into meaningful numerical representations using static and contextual word embedding algorithms such as Word2Vec, BERT, and DistilBERT pretrained models to generate word embeddings. The word embeddings are thus fed into neural networks for the classification of log lines into designated labels. The experiments performed with the combination of DistilBERT embedding model and LSTM classifier network for logs generated from patient monitoring devices achieved an accuracy of 0.99 with macro-averaged precision of 0.96, recall of 0.93 and F1-score of 0.94 in a multi-label classification. The results showed promising signs towards the automation of log analysis of syslogs generated from test-builds and patient monitoring systems.

Keywords Log Analysis , Anomaly Detection , Elastic Stack , Word Embeddings , Machine Learning, Natural Language Processing

Preface

This thesis is my documentation of my one-year Master's thesis research carried out at GE Healthcare, Helsinki. I am deeply grateful to GE Healthcare Finland Oy and my line manager, Jason Lee, for providing this remarkable opportunity. I would like to thank my supervisor from Aalto University Prof. Paavo Alku for supervising my thesis work. I owe a great deal of gratitude to my advisors, Kaveh Samiee, who directed me in implementing plans and methodologies during the course of my research, and Meng Sun, who ensured that my thesis was going in the right direction.

I am forever grateful to my parents for their immeasurable love and support, without whom I wouldn't be where I am today. I am also thankful to my siblings who have continuously motivated me throughout my life, and to my friends for their unwavering support.

Vantaa, 9 July 2023

Kamal Dhakal

Contents

List of Tables

List of Figures

Symbols and abbreviations

ELK	Elasticsearch Logstash Kibana
ILM	Index Lifecycle Management
AI/ML	Artificial Intelligence/Machine Learning
NLP	Natural Language Processing
DL	Deep Learning
LLM	Large Language Models
NN	Neural Network
ANN	Artificial Neural Network
FFNN	Feed-Forward Neural Network
RNN	Recurrent Neural Network
MLP	Multi Layer Perceptron
LSTM	Long Short Term Memory
SGD	Stochastic Gradient Descent
CBOW	Continuous Bag of Words
BERT	Bidirectional Encoder Representation from Transformers
DistilBERT	Distilled BERT
GPT	Generative Pretrained Transformers
ELMo	Embeddings from Language Model
OOV	Out-of-vocabulary
CM	Confusion Matrix
SOTA	State-of-the-art
XML	Extensible Markup Language
JSON	JavaScript Object Notation
REST	Representational State Transfer
API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
RAM	Random Access Memory
EDA	Exploratory Data Analysis
CI/CD	Continuous Integration/Continuous Deployment
GB	GigaBytes
CLI	Command Line Interface
SQL	Structured Query Language
KQL	Kibana Query Language
ILM	Index Lifecycle Management
HTTPS	Hyper-Text Transfer Protocol
TLS	Transport Layer Security
ETL	Extract, Transform Load
UI	User Interface

1 Introduction

Logs are a diagnostic resource to record runtime information about processes, including critical events executed within software systems. These logs help developers and product owners to understand system behaviour, monitor system health, and resolve various problems and anomalies that already exist or that could potentially arise in the systems. With the increase in complexity and ever-growing scale of systems, the volume of generated logs has been growing exponentially [1]. This explosive growth in logs has resulted in a corresponding exponential rise in anomaly cases. This scenario underscores the urgent need for effective log management solutions along with effective log analysis and log anomaly detection in the industrial landscape.

The process of log management and analysis includes the practice of continuously gathering, storing, processing, synthesizing, and analyzing log data and its distribution collected from disparate sources. This helps to optimize system performance, identify technical issues, better manage resources, strengthen security, and improve compliance. Historically, log analysis has been conducted by software developers, testers, or system administrators by manually reviewing the log lines of individual log files. This manual analysis involves simple keyword searches such as "failed", "error", "exception" or some rule-based methods to locate suspicious logs that might be associated with system problems. However, as the volume of logs is growing exponentially, manual approaches are inefficient in terms of time and resources. Moreover, simple keyword searches are susceptible to errors, so they cannot completely grasp the sequence of events that leads to an anomalous log event.

This thesis mainly focuses on the syslogs generated from patient monitoring systems in GE (General Electric) Healthcare. Currently, these logs are analysed manually in the company with keyword searches and rule-based filters which is tiresome and inefficient. Thus, there is a need for a better solution for the process of log analysis and anomaly detection in the company. With the recent advances in machine learning and natural language processing (NLP), the nature of log analysis has shifted towards data-driven approaches. However, treating logs as natural language has its own challenges because of the semi-structured or unstructured nature of log data generated in large volumes. Logging statements have a significant proportion of high-cardinality fields which adds complexity to the application of NLP techniques to log data. High-cardinality field refers to the unique field or data such as timestamps and Internet Protocol (IP) address. Despite the challenges, many studies have been conducted to investigate the benefits of recent NLP techniques with machine learning to improve the results of detecting anomalies in logs [2, 3, 4, 5]. This thesis expands on previous studies on log anomaly detection by applying machine learning and NLP techniques to real-world log data in an industrial context.

1.1 Objective and Research Goals

The objective of this thesis is twofold: 1) To design and implement a log management system using the Elastic Stack for exploratory data analysis, and 2) To employ machine learning and natural language processing techniques to analyze and detect anomalous log events. Based on these objectives, the thesis aims to answer the following research questions:

RQ1 How can a log management tool be configured to aggregate logs from multiple sources into a central analytics engine for visualizing numerical insights of the log data?

RQ2 How can the logs be preprocessed such that meaningful information can be extracted by removing high-cardinality fields in log data?

RQ3 What kinds of word embedding models can be used for effective representation of log data? How can these word embeddings derived from log data be leveraged for downstream tasks of log analysis and anomaly detection?

RQ4 What kinds of machine learning classifiers are effective for log anomaly detection which are robust to real-world log data generated from ambulatory monitoring, test builds, and clinical devices?

RQ5 As large language models based on neural networks are computation-hungry and demand large computational resources, how can a trained model be deployed in a resource-constrained environment?

To achieve the objectives of this research, an exploratory data analysis tool was developed using the Elastic Stack. This helped to obtain numerical insights of log data in large volumes before applying machine learning solutions to the problem. Following this, NLP techniques for the analysis of logs and anomaly detection were implemented. Various word embedding techniques were explored to convert text in log files into numerical representations, and subsequently, machine learning classifiers were applied. To support this process, a dataset was prepared involving the labeling of logs into several distinct classes. This comprehensive approach ensured a robust and systematic analysis of the log data leading to the effective log analysis and anomaly detection in log files.

1.2 Structure of the Thesis

This thesis is organised and structured into 7 chapters. Chapter 2 presents the overview of the log management system which is composed of Elastic stack, which serves as an exploratory data analysis tool. Chapter 3 presents the overview of the theoretical details of tokenization and word embedding techniques implemented in this research. Chapter 4 walks through the theoretical concepts of machine learning algorithms used in the area of NLP along with insights on model deployment. Chapter 5 gives the implementation and experimental details of generating word embeddings and implementing machine learning classifiers for the purpose of anomaly detection. All experimental results conducted in this thesis, along with their evaluations, are presented in Chapter 6. The findings of the thesis are summarised and concluded in Chapter 7.

2 Log Management and Exploratory Data Analysis

This chapter provides a theoretical overview of logs, supplemented by examples from logs generated in GE Healthcare. It describes the different formats of logging statements practiced in the company. It provides a high-level overview of a log management tool, Elastic Stack and its components, Elasticsearch and Kibana. The log shipping and processing tool, Elastic Agent is also described in this chapter.

System logs and their structures and the logging platform in GE Healthcare are presented in Section 2.1.1. Section 2.2 covers the log management tools and current approaches about log management platforms in the industry. Elastic Stack is broadly covered in this chapter along with shipping tools and mechanisms. Section 2.2.1 gives an overview of Elasticsearch and gives a detailed description of implementation steps. Section 2.2.2 discusses a visualization tool "Kibana" along with the implementation details. Section 2.2.3 gives in-depth information about Elastic Agent along with its implementation details. Within this section, the mechanisms by which Elastic Agent collects data from various sources and processes for transforming data prior to its indexing into Elasticsearch are explained. Section 2.4 presents how Elastic Stack with its search and analytical features helps to explore the data and its distribution prior to the application of machine learning algorithms.

2.1 System Log and Structure

Log data is a chronological sequence of single or multi-line events generated by software systems to permanently capture specific system states, and run-time information for diagnostic analysis in the events of certain failures or unexpected incidents [5]. Log data is generally text data stored in files with detailed run-time information about the system. Log data ranges from structured vectors (comma-separated values) over semi-structured objects (key-value pairs) to unstructured human-readable messages with heterogeneous event types. Generally, any well-established software or an operating system service writes sufficient log data, but a team of software engineers developing new software has to take into account that logging does not come automatically and could be executed in many different ways. General understanding while writing log statements is mixing different formats should be avoided and writing unstructured logs is discouraged as it makes the debugging unnecessarily complex and inefficient.

Generally, log data is generated in a sequential order in the form of log lines and each single line, commonly known as log message is a result of print statements placed as a form of logging statement in a source code. These print statements are also called logging statements placed purposefully by software developers throughout their code-base to support understanding of program activities and debugging. These print statements consist of static parts i.e. hard-coded strings and variable parts i.e. parameters that are dynamically determined during run-time. The single log line gives information about the system at one particular point in time as surrounding logs in the group give a dynamic workflow of the underlying program logic. Although there is no standard format for placing logging statements, each log message is at least expected to have some common information, such as timestamp, device name where it is generated, log label, and a free text message with information about the system. However, in practice, software developers sometimes knowingly or unknowingly implement logging systems in a free-style format according to their convenience without any common parts. Also, these print statements could change a lot when the source code is updated and sometimes totally new kinds of log entries are added when the software is updated. Although log messages are typically contained in a single line in text format, it is also possible that log messages span multiple lines and are represented in other formats such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON) [6].

These kinds of log data are stored as files on a disk. This approach allows to appending of new messages at the end of the file resulting in a file containing log messages in chronological order. There are also other methods of saving these log data such as storing them in a database or transmitting it directly into a log collector solution that could be part of a central log management system. Software logs are especially useful in debugging while the software is still under development and for troubleshooting when the software is actively used [6]. System administrators are one of the heaviest log file users as logs can be used to monitor system health and detect undesired behaviour. Log data can also be used for many other purposes such as profiling and benchmarking the software, and it can provide various possibilities such as user behaviour analysis.

With the increase in software complexity, the amount of generated log data has seen an unprecedented level. The amount of logs generated each hour has surpassed tens of gigabytes in volume spanning 100s of millions of lines in large corporations. This is where an effective log management system and automated log analysis become a necessity.

2.1.1 Syslogs in GE Healthcare

This section presents the type of syslogs generated in GE Healthcare which are the main source of data used in this research work. This section describes the log sources in GE Healthcare, the formats that are commonly practiced during logging statements, and protocols used in saving those log lines into a logging system. This section also presents different examples of log data generated across different systems.

Figure 1 shows the example of a log message that follows the standard syslog format used in GE Healthcare generated by the system during nightly tests built in wearable devices. This log message is built on a format where *'<135>'* is facility+priority, *'1'* is a version, *'2022-11-18T10 :06:59.115439+00:00'* is timestamp, *'qemux86'* is hostname, *'axone_comm'* is application name, *'765'* is process id and rest of the text is a free text message. The hostname *'qemux86'* here is a virtual hub from where this syslog was generated and this plays an important role in the analysis process. The use of the field 'facility+priority' is not always prevalent across all syslogs and those kinds of syslogs are documented as we walk through the examples.

```
<135>1 2022-11-18T10:06:59.115439+00:00 qemux86 axone_comm 765 - -
    0|main|337|PatientRegistryIpc.cpp:82|[PatientRegistryIpc] Patient
     registry changed: { "PatientAssignmentState": NOT_ASSIGNED, "
    Patient": , "source": AXONE }
```

Figure 1: Example of syslog generated during nightly test build

Figure 2 shows three syslog messages typically in the same format as in the above example in Figure 1. In the examples, we can see hostname as *'44-4b-5d-00-01-99'* which is a physical hub device. The log messages are generated in a chronological sequence in a log file. We can see some errors in the example log lines which are interesting regions in log analysis both manual inspection and automated anomaly detection using machine learning. The log examples presented show the abundance prevalence of inconsistent use of whitespaces and special characters.

```
<134>1 2022-11-22T01:12:45.118006+02:00 44-4b-5d-00-01-99
    axone_logger 2278 - -  1669072365.111 (5) axone_logger
    [2278/1968621488] <L> 0|main|19|Main.cpp:334|Exiting
<131>1 2022-11-22T01:12:48.047901+02:00 44-4b-5d-00-01-99 axone_comm
     2274 - -  0|1960809376|11|RegistryDataAccessor.cpp:33|
    getRegistryDataList failure: Error return: Query timed out.
    Partial results may still be available.
<131>1 2022-11-22T01:12:48.048607+02:00 44-4b-5d-00-01-99 axone_comm
     2274 - -   (ERROR) axone_comm[2274/1960809376] <L>
    0|1960809376|11|RegistryDataAccessor.cpp:33|getRegistryDataList
    failure: Error return: Query timed out. Partial results may still
     be available.
```

Figure 2: Example of syslog generated from Hub

Figure 3 shows another example of a log message that is generated by the patient monitoring system. The host-name *"DESKTOP-0TVE4F"* is a viewer platform. In the log message presented in Figure 3, log level "WARN" is present giving a clear indication of something happening within the system or patient enrolled with the system. This label is important to diagnose the system during the inspection process. It can be noticed that 'facility+priority' and 'version' are not included in this log format and the rest of the log message follows a similar format as in examples shown in Figure 1 and 2.

```
"2023-04-10T19:48:40.232+0000","5632","5632","WARN","DESKTOP-I0TVE4F
    ","744d9ad9-727c-4cb9-8d9c-6545d9522c32","","Solarflare
    [27820:113]","com.ge.hc.lcs.axone.solarflare.source.a.a:234","
    General","umfnum sample lost: SampleLostStatus[total_count=7685,
    total_count_change=30, last_reason=LOST_BY_WRITER]"
```

Figure 3: Example of syslog generated from patient monitoring viewer platform

With the introduction of log levels, it is important to know the types of log levels that are used in logging statements practiced in the company. The message field in the logging statements "level" is restricted to 8 types, namely *'CRITICAL'*, *ERROR*, *WARNING*, *NOTICE*, *INFO*, *DEBUG*, *TRACE*. Here, we give a brief glossary of the log levels typically exercised in logging statements by software developers in GE Healthcare.

CRITICAL: Indicates an anomalous condition from which a service or application cannot recover. For example, a non-recoverable error causes the service or application to stop working.

ERROR: Indicates anomalous condition in which a specific function of a service or application cannot recover. For example, failure to read configuration settings; however, default settings can be used.

WARNING: Indicates a potential condition, however, it does not cause the service or application to stop functioning. For example, a loss of expected network packets a loss of connectivity, or a docker container unexpectedly restarted.

NOTICE: Indicates a normal condition but significant condition. For example, a persistent storage is 90 percent storage full, or the Network Time Protocol (NTP) configuration was changed.

DEBUG : Useful for developer-related info. A log message at this level tends to be useful only to software developers for debugging. Its use is discouraged in production environments and should be turned off by default.

INFO: Indicates a normal event or condition that does not require intervention and is not anomalous. For example- logging of performance Metrics.

TRACE: Useful for developer-related info, including high resolution and potentially high overhead traces. A log message at this level tends to be useful only to software developers for debugging. When possible, production software is built with the TRACE level compiled out (language-dependent feature). Trace should not be enabled in a production environment.

Figure 4 shows the snippet of another log message. This log is generated from the hospital bed where the patient is assigned to devices with respiration and pulse oximetry. It can be seen that the protocols and association state machines in the log example. Protocol "R" means the respiration sensor and "Association state": "ASSOCIATED_NO_BATTERY" means the respiration sensor has lost connection to the battery. As we can see this log message also does not include the field 'facility+priority' but all other format remains the same as in previous examples. The length of this message is comparatively longer than that of regular log messages.

```
2022-10-15T15:28:03.159+0000 44-4b-5d-01-04-89 data 586
    AssociationManager.cpp 807 Sensor state changed: {"Association key
    ": "0x444b5dff01000f58", "Pairing key": "0xfe043895cae56480", "
    Connection key": "0xe2", "Client address": "0xe2", "isAlive": "1",
     "Overtaken": "0", "Nodes": [{"Address": "0x26", "Bandwidth": 3, "
    Capabilities": [{"Protocol": "A", "Version": "0", "Parameters":
    ""}, {"Protocol": "b", "Version": "0", "Parameters": "0
    x3cef0a640fed014a013c014c013dffff06bc0007071800fff3ffffffff00005c0000
    "}], "Sw version": ":tsb:mPuck:v1.0.4.1.1.1-1183.1.207efffa.master
    .release"}, {"Address": "0x27", "Bandwidth": 8, "Capabilities":
    [{"Protocol": "R", "Version": "0", "Parameters": "0x0101"}], "Sw
    version": ":tse:mResp:v1.0.4.1.1.2-1068.1.68181520.master.release
    "}], "Association state": "ASSOCIATED_NO_BATTERY"
```

Figure 4: Syslog example generated from ambulatory monitoring sensor platform in hospital settings

2.2 Elastic Stack

Elastic Stack is best described as a stack consisting of a group of four product projects from Elastic, namely, Elastisearch, Logstash, Kibana, and Beats. It was formerly called ELK stack, consisting of only three sets of components - Elasticsearch, Kibana, and Logstash. After the addition of the fourth component, Beats, the ELK stack evolved into a complete stack, making it an Elastic Stack. All the components of the Elastic Stack are designed to work together, enabling users to securely take data from any source, in any format, and then search, analyse, and visualise it [7]. Elastic, as a company develops and maintains tools and shares many similarities for configuration of directory layouts with their own configuration represented in ".yml" files. Although these components are often used together, each component can be used independently or integrated with other systems depending on the specific requirements of a project. Moreover, learning one component of the Elastic Stack makes it easier to learn other components as they share commonalities in design philosophy, terminologies and technical workflow.

Elastic Stack was used as a tool for exploratory data analysis in this research. By executing a variety of search operations using Kibana Query Language (KQL), we were able to obtain the statistical distribution and numerical insights of log data. This also acted as a centralised logging system as logs from different sources were dumped into a central unit, Elastic Stack. This was used as a complementary tool before the application of machine learning for log analysis. This process was vital in gaining a deeper understanding of the data and its underlying statistical features.

As there were multiple sources of syslogs in the company, it was necessary to build custom log shippers to Elasticsearch which could integrate multiple sources. Elastic Agent with custom log integration was used as a data shipper to ingest data to Elasticsearch. The analytics performed with Elasticsearch were visualized in Kibana in a browser. It was discovered that the Elastic Stack can serve as an effective log management tool where logs can be analysed as a central logging unit. This section gives detailed information about Elasticsearch and Kibana along with Elastic Agent as these components are extensively used in this project. Moreover, this chapter also presents the implementation details and configuration settings of components used in Elastic Stack. This section presented detailed information involving the implementation details and challenges that arose during the process. This section also gives insights on target log data along with parsing and indexing into Elasticsearch. The section follows the development process from the initial plan to the deployment of Elastic Stack in a sequential order. The system design of Elastic Stack is shown in Figure 5.

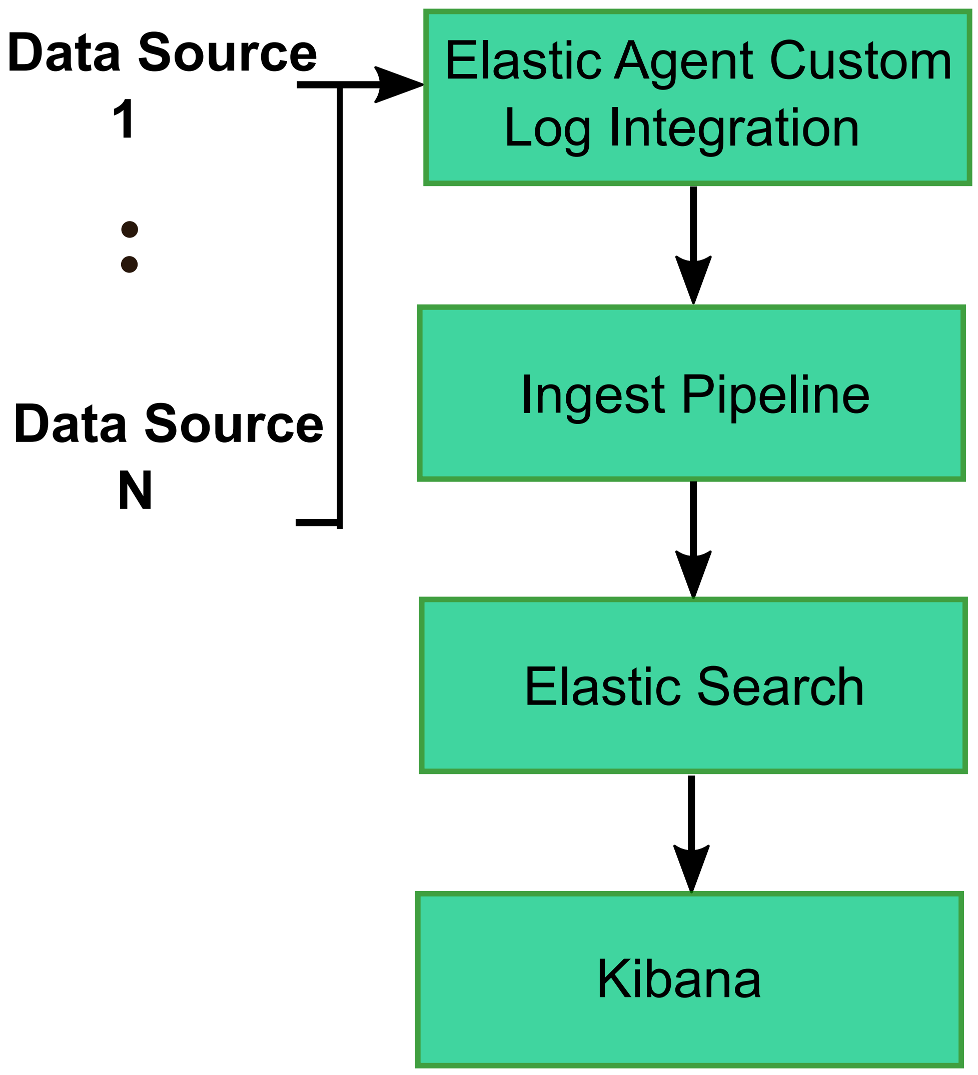


Figure 5: Design framework of the implemented Elastic Stack for log management and exploratory data analysis

2.2.1 Elasticsearch

Elasticsearch is a distributed search and analytical engine that is built on Lucene, an open-source full-text search library. It was first released in 2010 and has been the core component of the Elastic Stack. It was operated as an open-source project with Apache Licence 2.0 since its creation in 2011 until January 2021. When the company moved from the ELK stack to Elastic Stack along with the major release of 8.0.0 version, Elastic, as a company changed the licensing model of Elasticsearch and Kibana from open-source to a dual licensing model under Server Side Public Licence (SSPL) and Elastic Licence, which are not recognised as open-source licenses by the Open Source Initiative (OSI). This move was a response to what Elastic saw as a threat from cloud service providers, particularly Amazon Web Services (AWS), offering Elasticsearch and Kibana as services without contributing back to the open-source project or collaborating with Elastic. This change in licensing sparked a lot of controversy in the open-source community. In response, AWS, along with other partners, launched OpenSearch, a community-driven, open-source fork of Elasticsearch and Kibana, which continues to be licensed under Apache 2.0.

As Elasticsearch is built on Apache Lucene, all the search operations are carried out by Lucene. However, Elasticsearch provides additional functionality and a more user-friendly interface to Lucene's powerful capabilities, making it easier to use and manage. Elasticsearch simplifies using Lucene by providing REST (Representational State Transfer) API (Application Programming Interface) making it easier to perform operations in many programming languages. This is simplified compared to Lucene where direct Java code is needed to interact with the library. Elasticsearch uses a flexible data model, accepting JSON documents for storage and indexing. This is much easier to work with compared to Lucene's more rigid data structure. Also, Elasticsearch provides many additional features out of the box, including support for multi-tenancy, an integrated analytics engine, and features for security, monitoring, alerting, etc. [8]

Elasticsearch is designed to operate in a distributed and scalable manner. This design allows multiple instances (nodes) of Elasticsearch to work together as a cluster to efficiently handle API calls and provide high performance for search and data retrieval operations. In Elasticsearch terms, each instance is called a node, and a group of instances is called a cluster where each node has different roles. Each cluster is assigned with at least one elected master node which is responsible for cluster-wide changes such as creating and deleting an index. The master node oversees cluster-wide modifications like index creation/deletion and node addition/removal, as well as load balancing. If it fails, another master-eligible node takes its place. Ingest nodes prepare incoming documents for indexing by applying transformations, then return them to their origin for actual indexing. Data nodes store indexed data and respond to queries. It's generally recommended to allocate dedicated nodes for each role within a cluster.

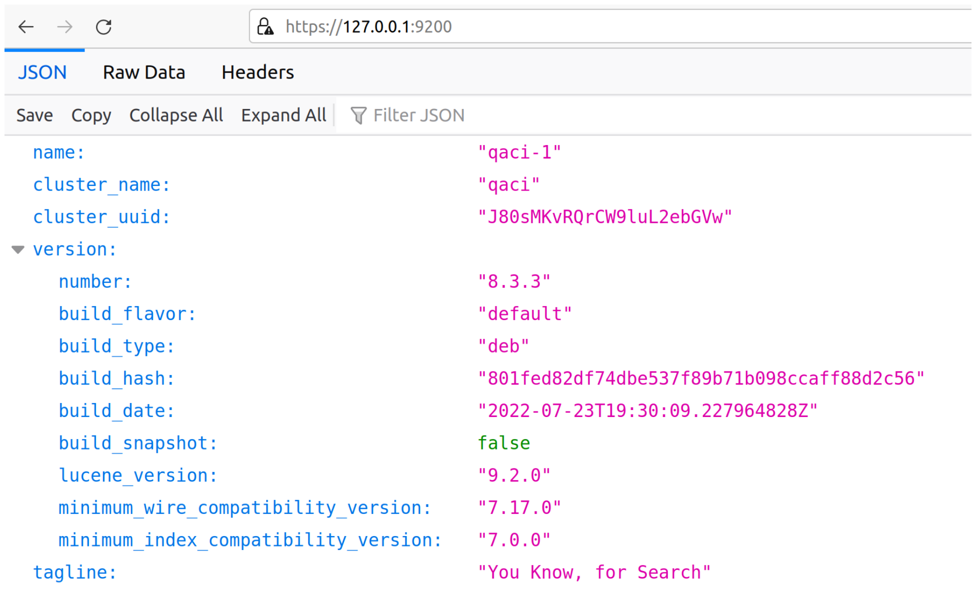
Elasticsearch, a distributed JSON document store, has some similarities to relational databases but operates differently due to its use of an inverted index and other data structures. Unlike relational databases that store data in rows and columns, Elasticsearch indexes serialized JSON documents, making them fully searchable in real time. An Elasticsearch index is a collection of documents that are related to each other. Each document correlates a set of keys (name of fields and properties) with their corresponding values (string, numbers, Booleans, dates, arrays of values, geolocations, and other kinds of data). Elasticsearch uses a data structure called an inverted index, which is designed to allow very fast full-text searches. An inverted index lists every unique word that appears in any document and identifies all the documents where each word occurs.

Elasticsearch divides its indices into shards, each being a Lucene index that contains a portion of the Elasticsearch index's documents. Each Lucene index can store up to approximately 2.1 billion documents, due to limitations of a 32-bit signed integer. Furthermore, these Lucene indices are split into smaller parts known as Lucene segments. Lucene segments, generated when new documents are indexed, are immutable once created. Though Elasticsearch can merge smaller Lucene segments, either automatically or manually via API. Fewer segments enhance efficiency because Lucene processes segments sequentially, not in parallel. Therefore, a larger number of segments can slow down the search performance.

Elasticsearch communication is done by a comprehensive REST API that uses HTTP requests (GET, PUT, POST, DELETE) for data operations, paralleling CRUD (Create, Read, Update, Delete) operations in databases. Interactions can be done via the Kibana console, cURL on a command line, or other clients. For instance, entering "GET /_cat/indices" in the Kibana console retrieves information about existing Elasticsearch indices using the cat API. Similarly, if Elasticsearch is hosted locally, the command would be curl -XGET http://localhost:9200/_cat/indices.

Installation of Elasticsearch

Elasticsearch is free and open to use and has clear documentation on installation and configuration procedures. However, there are some caveats in the process. The firewall and proxies in the corporate environment make it painful to access the required ports and network configuration. The Elasticsearch version of 8.3.0 was installed which was the latest version during the start of this research project. The computer was a standalone system with Ubuntu 22.4 LTS as the operating system and with a Random Access Memory of 32 GB. The steps were followed according to installation guidelines in the documentation with some customization done in the allocation of memory. As Elasticsearch installation defaults to allocating 1 Gigabytes of RAM dedicated to Elasticsearch, it was necessary to allocate more memory as we had more computationally demanding tasks. So, for this Elasticsearch.yml file was amended to increase the allocated memory to 8 Gigabytes. Elasticsearch was configured in localhost however, but Elasticsearch provides the customization feature which can be implemented in *"Elasticsearch.yml"* file to operate it over the network in other system browsers or command line interface (CLI) through the terminal. After the installation of Elasticsearch, it was tested using the curl command and also through the browser to verify the installation. The result from the browser using Universal Resource Locator (URL) on port number 9200 is shown in Figure 6.



```
name:                                   "qaci-1"
cluster_name:                           "qaci"
cluster_uuid:                           "J80sMKvRQrCW9luL2ebGVw"
version:
    number:                             "8.3.3"
    build_flavor:                       "default"
    build_type:                         "deb"
    build_hash:                         "801fed82df74dbe537f89b71b098ccaff88d2c56"
    build_date:                         "2022-07-23T19:30:09.227964828Z"
    build_snapshot:                     false
    lucene_version:                     "9.2.0"
    minimum_wire_compatibility_version: "7.17.0"
    minimum_index_compatibility_version: "7.0.0"
tagline:                                "You Know, for Search"
```

Figure 6: Installation verification of Elasticsearch in browser with localhost and specified port number

2.2.2 Kibana

Kibana is a visualization tool for the Elastic Stack with a friendly user interface. It was developed and designed by Rashid Khan in 2012 and it quickly became an integral part of the Elastic Stack together with Elasticsearch and Logstash [9]. Kibana is tightly integrated with Elasticsearch and the larger Elastic Stack and this makes it ideal for searching, viewing, and visualizing data indexed in Elasticsearch. It also makes an ideal platform to analyze data through the creation of bar charts, pie charts, tables, histograms, and maps. A dashboard view assembles these visual elements, which can be shared via a web browser to provide real-time analytical views into large data volumes to various use cases. Kibana is also used to monitor, manage, and secure Elastic Stack instances easily through a web interface. Kibana gives centralized access to Elastic Stack for observability, security, and enterprise search applications.

Data visualization in Kibana can be carried out from an Elastic index or multiple Elasticsearch indices. The analytics section in Kibana, known as Kibana Analytics, is used for data exploration, visualization, and analysis. It incorporates the "Discover" tool that serves as an interface to the data stored in Elasticsearch. By default, it utilizes KQL for search queries and filtering, although Lucene query syntax is an alternative. The queries can be personalized by selecting specific fields to display, which can then be saved and integrated into a dashboard. After the queries are carried out, the results can be visualized through standard chart options or built-in chart options like Lens, Canvas, and Maps. Placing these "Discover" windows on a dashboard allows for a useful display of varied data types from multiple indices simultaneously. The dashboard tool facilitates the creation and administration of all these dashboards.

Moreover, Kibana provides a feature for exploring the context around specific documents or log entries. This 'surrounding documents' functionality is particularly useful when users need to understand the conditions or events happening around a specific log entry or indexed document. The resulting data views can be saved and added to a customized dashboard. These dashboards can host multiple 'Discover' windows, offering a flexible way to display diverse types of data from different indices in a consolidated view. The comprehensive dashboard tool facilitates the creation, customization, and management of these insightful visual displays, making data analysis more efficient and intuitive.

The Visualize Library in Kibana enables the creation of varied styles of visualizations, including pie charts, gauges, line graphs, bar graphs, tag clouds, and more. The tools section permits the incorporation of text, images, dropdown menus, and range sliders into dashboards. These controls can adjust the filters within a dashboard. For instance, a dropdown menu can be set up to display all values of a specific field in an index. Selecting a value from this menu triggers a filter that updates all the views within the dashboard, including 'Discover' objects, to display documents where the chosen value appears within that field. However, to include text fields in the dropdown menu, they must be indexed as keywords since the options are dynamically generated using terms aggregation, which is only possible when text fields are indexed as keywords.

Enterprise Search in Kibana provides tools to create, deploy, and manage search experiences for websites and mobile applications, and to execute search queries across connected platforms. The "Observability" section offers tools for monitoring metrics, log collection, and alert generation. The Security section provides tools related to Elasticsearch cluster security. While these components of Kibana are valuable and offer a wide range of utilities, they are beyond the scope of this project as a Proof of Concept (PoC) and hence, aren't utilized.

Installation of Kibana

After the successful installation of Elasticsearch, visualization tool, Kibana installed following the guidelines in elastic documentation [10]. The configuration file *"kibana.yml"* was updated to specify the port details and required Internet Protocol (IP) address and Domain Name System (DNS) configuration. The process itself was not difficult but the corporate proxies and firewall along with CA certificates made the installation process difficult. As Kibana is tightly integrated with Elasticsearch, it was necessary to match the configurations in *"elastisearch.yml"* settings to the *"kibana.yml"*. As Kibana runs on a browser, the security over the Hyper Text Transfer Protocol (HTTPS) and Transport Layer Security (TLS) was configured following the documentation [11] provided by Elastic. The login window of Kibana with an integrated Elasticsearch in the backend is shown in Figure 7.

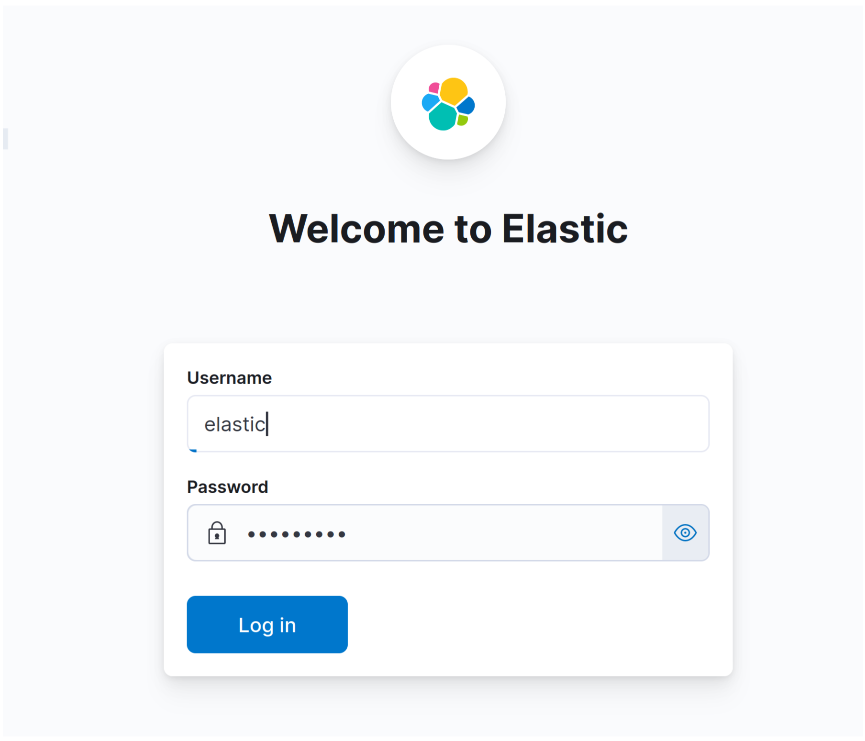


Figure 7: Kibana login window with Elasticsearch integrated in the backend

2.2.3 Elastic Agent

Elastic Agent is a single, unified way to add monitoring for logs, metrics, and other types of data to a host. It serves as a safeguard against security risks, can request data from operating systems, and transmit data from distant services or hardware among other functions. A single agent makes the process of deploying monitoring across the whole infrastructure becomes quicker and easier. Each Agent comes up with a unique policy that supports the integrations to be added for the new data sources security protections, and so on [12].

Elastic Agent and Beats processors enable the users to manipulate the data at the edge. This becomes particularly useful if the user needs to control what data is sent across the wire, or needs to enrich the raw data with information available on the host. This data manipulation is facilitated by the Elasticsearch ingest pipeline as the processing of data is done as it arrives on the fly. This prevents additional processing overhead on the hosts from which the data is being collected.

Preprocessing logs before indexing data into Elasticsearch is done usually by processors using the ingest pipeline to apply transformations in the incoming data. Some examples of transformations are adding or removing fields, extracting values from text, and using filters such as grok, dissect, etc. The processors are stacked in series in a pipeline and they operate in a sequential manner making specific changes to incoming documents. After the operation of processors is completed, Elasticsearch finally performs the indexing of transformed documents. Ingest pipelines can be created using the ingest pipelines in Kibana or using ingest APIs in elastic [13]. The ingest pipeline is shown in Figure 8 which shows that the incoming documents are transformed through a series of processors before indexing to Elasticsearch. Figure 8 shows one example of a Dissect processor, however, multiple processors can be stacked in series.

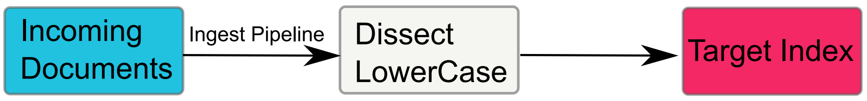


Figure 8: Ingest Pipeline with data transformation processors for transforming data before indexing to Elasticsearch

As demonstrated in Figure 9, the Elastic Agent has the capability to monitor the host where it is deployed, as well as collect and relay data from remote services and hardware where direct installation is not feasible.

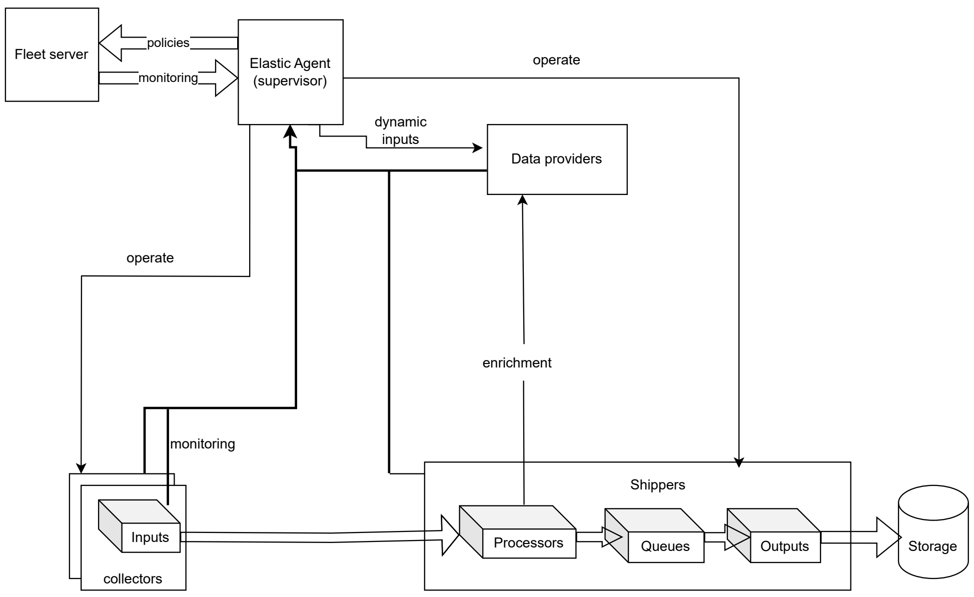


Figure 9: Elastic Agent architecture with fleet server deployed in host machine

Elastic integrations provide an easy way to connect Elastic to external services and systems and quickly get insights or take action. These integrations are designed to collect data from novel sources and typically come prepackaged with readily usable resources such as dashboards, visualizations, and pipelines. These pipelines are beneficial in extracting structured fields from logs and events. The process of insights is therefore significantly seamless, often providing useful information within a matter of seconds. Elastic has made available integrations for a variety of popular platforms and services like AWS, Ngnix, in addition to several generic input types such as log files, making it versatile for many use cases. Here in this thesis, we use custom log integration with Elastic Agent for shipping logs to Elasticsearch.

Kibana offers a user-friendly interface that allows users to add and manage these integrations with little effort. One of the standout features of Kibana is the unified view it provides of the available integrations, showcasing both Elastic Agent and Beats integration in one place. This unified view simplifies the process of selecting and implementing integrations, providing a clear overview of all the options at a glance. Users can conveniently browse through the available integrations, compare their features, and select the most suitable ones for their specific needs. This approach not only reduces the time and effort involved in managing integrations but also optimizes their performance by ensuring they are well-aligned with the user's objectives and system requirements. In addition, the unified view is continuously updated, ensuring that users have access to the latest Elastic Agent and Beats integrations as soon as they become available. This ensures that users can always take advantage of the latest features and improvements to enhance their system's performance and functionality. This streamlined integration management, combined with the power and versatility of Elastic integrations, ultimately enables users to maximize the value of their data and achieve their operational objectives more efficiently and effectively. Elastic Agent Policy is used to specify the choice of integrations that the user wants to run on the host that the user specifies. Elastic Agent policy can be applied to multiple agents, making it easier to manage configuration at scale.

Logs are append-only time series data occurring in a sequential manner with timestamps where successive data points are added at the end of the series. Append-only refers to the data structure's property where once data has been added, it is considered immutable, meaning existing data is never modified or deleted. This is highly useful for use cases such as logging and event tracking where it is critical to preserve the original data for analysis, monitoring, and debugging.

Indexing and search requests can be submitted directly to a data stream. The request is automatically routed to backing indices that store the stream's data. Index Lifecycle Management (ILM) can be used to automate the management of these backing indices. For example, ILM can be used to automatically move older backing indices to less expensive hardware and delete unneeded indices. ILM helps to reduce costs and overhead as the volume of data increases.

A data stream needs a matching index template. An index template is a way to tell an Elasticsearch how to configure an index at the time of indexing. For data streams, the index template configures the stream's backing indices as they are created. Templates are configured prior to index creation.

Installation of Elastic Agent

After the successful integration of Elasticsearch and Kibana, the installation was verified and we explored the features of this combined integration of Elasticsearch with Kibana. Kibana, with its robust user interface, provides the feature for where ingestion of single files up to 100MB via a simple drag-and-drop operation. However, this research work required a logging system where logs from a multitude of sources could be received by a single engine. Additionally, a prerequisite to the indexing of data into an Elasticsearch index was the parsing of the data as logs from different sources do not follow the same format or structure. To fulfill these requirements, the Elastic Stack is equipped with the provision of the Filebeat module, which acts as a lightweight data shipper. This module is used for the transportation of data from several sources to Elasticsearch or to Logstash to parse and transform data before indexing into Elasticsearch.

Logstash, another open-source tool, is frequently employed with Elasticsearch to transform data prior to indexing. It is essentially an open-source data processing pipeline that allows us to process logs and other event data from various sources. It is proficient in performing Extract, Transform, Load (ETL) transformations, which include extracting data from outside sources, transforming it to fit operational needs, and loading it into the end target (in this case, Elasticsearch).

However, with the introduction of of Elastic Agent presents a consolidated approach, capable of performing most tasks accomplished by the combination of Filebeat and Logstash. The Elastic Agent provides a unified way to add monitoring for logs, metrics, and other types of data from your hosts and services. This flexibility and ease of management are further enhanced with the Elastic Agent's integrations, significantly simplifying the data ingestion and transformation process.

The installation of Elastic Stack was carried out by following the Elastic documentation. Elastic Agent was installed on the same machine where Elasticsearch and Kibana were installed. The operating system of the system was 22.04 LTS Ubuntu. The system was equipped with 32 GB of Random Access Memory (RAM). Prerequisites of installing Elastic Agent are mentioned in [14]. One of the prerequisites is the installation of Fleet which is described below.

Installation of Fleet in Air-gapped environments

As this research was conducted in a corporate environment, Elastic Agent is run in a restricted network. Thus we had to perform extra steps to ensure Kibana was able to reach the elastic package registry to download package metadata and content. Similarly, it must be ensured that Elastic Agent must be able to download binaries during upgrades. The orderly documentation for upgrading all the components in an air-gapped environment can be found in the elastic documentation guide [15].

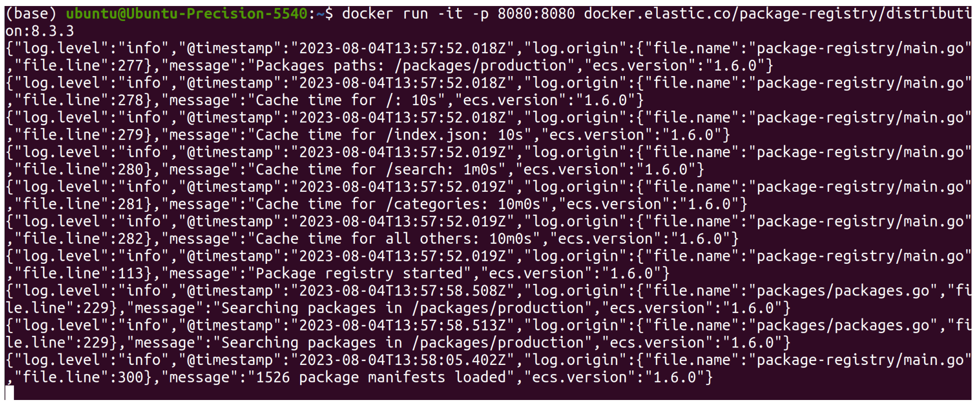
The Kibana downloads package metadata and content from the public Elastic Package Registry at `epr.elastic.co`. Thus, Kibana needs to have network access to connect to the elastic package registry. There were two options for that and those options are:

- Using a proxy server to access the Elastic Package Registry.

- Host own Elastic Package registry using Docker Image.

The routing of traffic through a proxy server did not materialize because of strict policies inside GE Healthcare. Thus, the Elastic Package Registry was deployed and hosted online using the available distribution of docker images. Every distribution contained packages that could be integrated with different versions of Elastic Stack and a suitable version of 8.3.3 docker image was installed. The steps for hosting the in-house elastic package registry in the docker image are given below:

- A docker image was pulled from the public docker registry.
- The pulled docker image was saved locally.
- "Elastic Package Registry" was run with a docker image. The instance of running the docker image with elastic package registry is shown in Figure in 10.
- Configuration file of Kibana `kibana.yml` was updated to connect Kibana to the in-house package registry. `xpack.fleet.registryUrl` property in the `kibana.yml` was set to the URL of the hosted package registry.
- File path to ca-certificates of the company was added to the Kibana startup file with `NODE_EXTRA_CA_CERTS` environment variable.



```
(base) ubuntu@Ubuntu-Precision-5540:~$ docker run -it -p 8080:8080 docker.elastic.co/package-registry/distributi
on:8.3.3
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.018Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":277},"message":"Packages paths: /packages/production","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.018Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":278},"message":"Cache time for /: 10s","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.018Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":279},"message":"Cache time for /index.json: 10s","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.019Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":280},"message":"Cache time for /search: 1m0s","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.019Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":281},"message":"Cache time for /categories: 10m0s","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.019Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":282},"message":"Cache time for all others: 10m0s","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:52.019Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":113},"message":"Package registry started","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:58.508Z","log.origin":{"file.name":"packages/packages.go","fi
le.line":229},"message":"Searching packages in /packages/production","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:57:58.513Z","log.origin":{"file.name":"packages/packages.go","fi
le.line":229},"message":"Searching packages in /packages/production","ecs.version":"1.6.0"}
{"log.level":"info","@timestamp":"2023-08-04T13:58:05.402Z","log.origin":{"file.name":"package-registry/main.go"
,"file.line":300},"message":"1526 package manifests loaded","ecs.version":"1.6.0"}
```

Figure 10: Docker image instance hosted with Elastic Package Registry

After the fleet server was configured and Kibana was accessed by the fleet server and integrations, Elastic Agent was installed and enrolled in the fleet. Installation steps are given below:

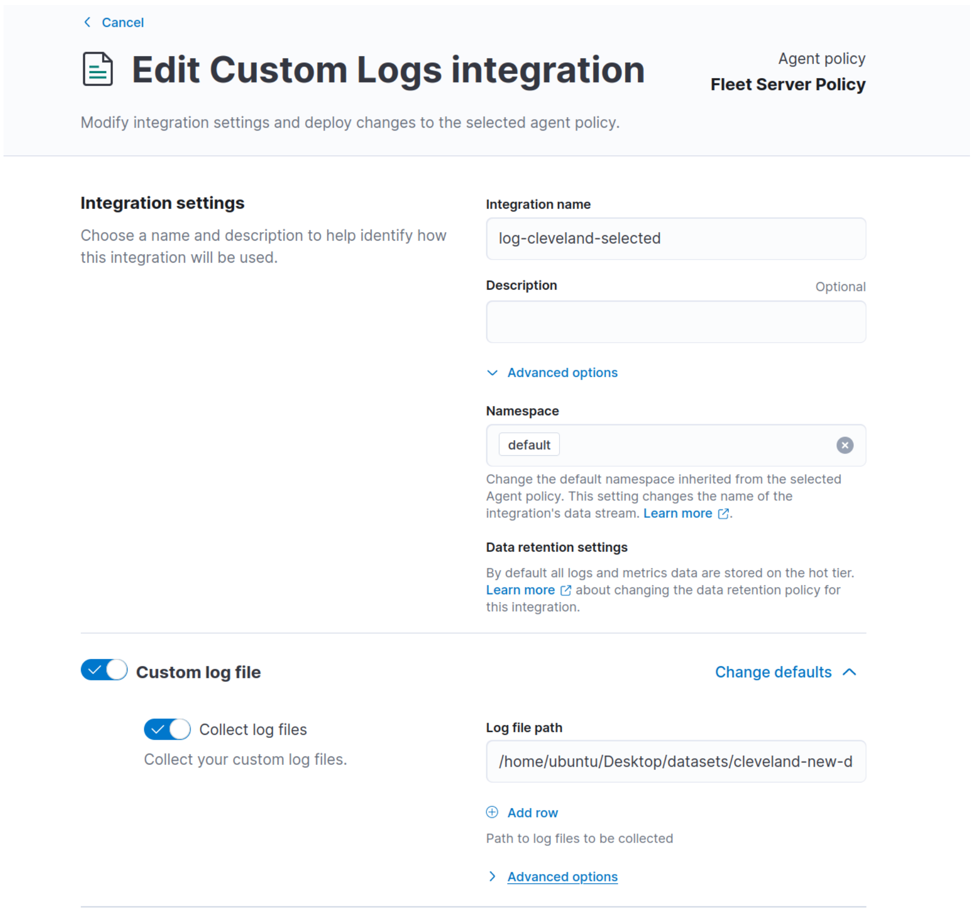
- Agent policy was created from the fleet screen in Kibana. Once the policy was created, the enrollment token was copied from the "Enrollment Tokens" tab in Fleet.
- Ingest Pipeline was created as logs data are custom data. Elastic Agent has default policies suitable for log files. So this step can be skipped if the log data is used for trivial analysis. This research was not only about Elastic

Stack so default policies also worked fine. However, the ingest pipeline was created for experimental purposes. Different processors such as grok, dissect, date, timestamp, etc. were used to transform the data before indexing to an Elasticsearch index. Ingest pipeline management UI allowed us to test custom pipelines with sample documents.

- Default Index Lifecycle Management (ILM) policy available in Elastic Stack was used for the experiments conducted in this thesis. However, it is advised to define its own ILM policies so that the system can be managed in accordance with the need for performance, scalability, and retention.

- Data stream was set up using a matching custom component template.

After this, custom logs integration was added to the Elastic Agent policy. The custom integration is shown in Figure 11.



Agent policy
Fleet Server Policy

Modify integration settings and deploy changes to the selected agent policy.

Integration settings

Choose a name and description to help identify how this integration will be used.

Integration name: log-cleveland-selected

Description (Optional):

Namespace: default

Change the default namespace inherited from the selected Agent policy. This setting changes the name of the integration's data stream. Learn more.

Data retention settings

By default all logs and metrics data are stored on the hot tier. Learn more about changing the data retention policy for this integration.

Custom log file ✓

Collect log files ✓

Collect your custom log files.

Log file path: /home/ubuntu/Desktop/datasets/cleveland-new-d

Path to log files to be collected

Figure 11: Elastic Agent integration with custom logs integration policy on a fleet server

2.3 Adding data to Elasticsearch

Elastic Stack provides several available options for getting data into Elasticsearch, which is known as ingesting or indexing the data. This project leveraged the Elastic Agent hosted in the fleet server with custom log integration with Elasticsearch and Kibana. The setup details of Elastic Agent are mentioned in the section above. This choice of use of Elastic Agent with custom log integration was made because the syslog data used in this thesis always followed a format with timestamps in each of the log lines.

The data ingestion pipeline was designed as when we send data directly into Elasticsearch. The data ingestion pipeline often includes additional steps to manipulate the data, ensure data integrity, or manage the data flow. It was necessary to sanitize, normalize, transform, or enrich log data before it was indexed or stored in Elasticsearch. The flow graph of implementing timestamped data in Elastic Agent is shown in Figure 12.

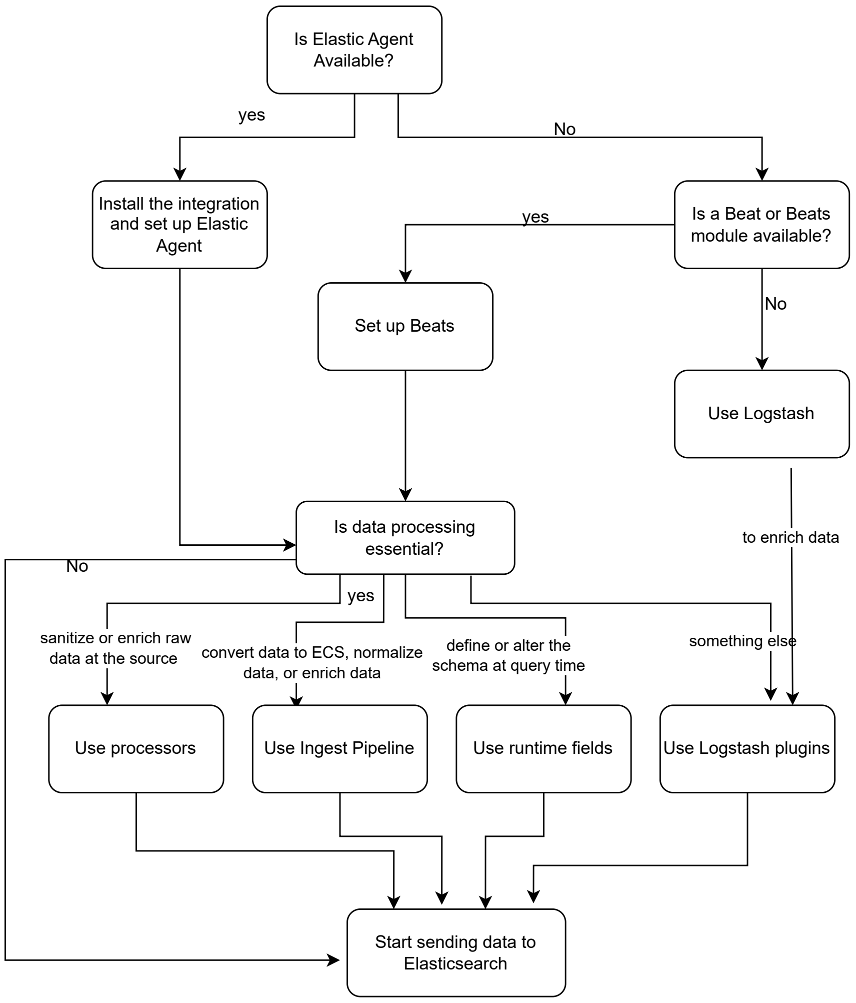
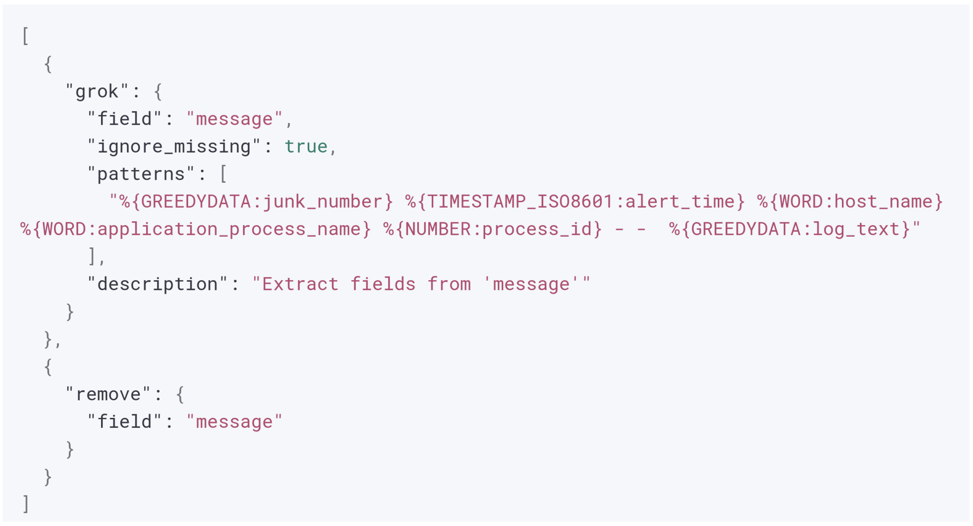

Is Elastic Agent Available?
yes
No
Install the integration and set up Elastic Agent
yes
Is a Beat or Beats module available?
Set up Beats
No
Use Logstash
Is data processing essential?
to enrich data
No
yes
sanitize or enrich raw data at the source
convert data to ECS, normalize data, or enrich data
define or alter the schema at query time
something else
Use processors
Use Ingest Pipeline
Use runtime fields
Use Logstash plugins
Start sending data to Elasticsearch

Figure 12: Data ingestion pipeline for timestamped data using Elastic Agent to be indexed into Elastisearch

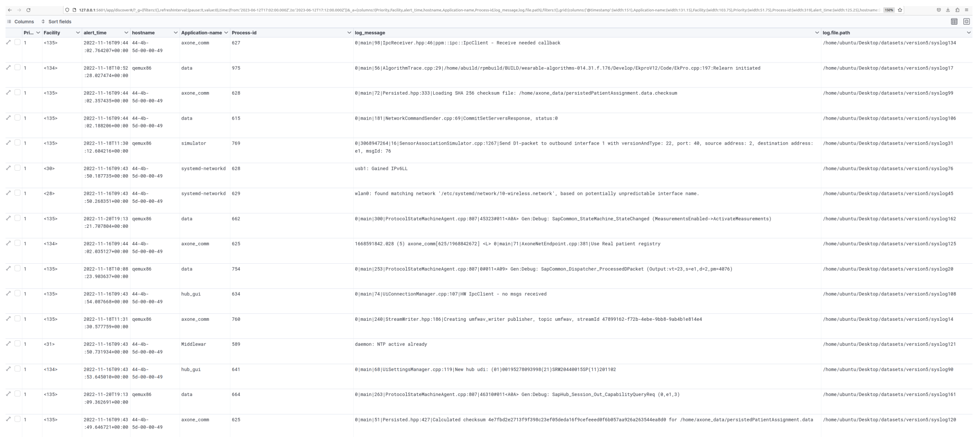
Here in this research, data ingestion to Elasticsearch is implemented using Elastic Agent and custom logs integration. Logs from multiple sources were seamlessly integrated using custom log integration using Elastic Agent. Log data source path was defined and it is modular to accommodate different sources. The ingest pipeline was defined to process our logs before they get imported to Elasticsearch. Ingest pipeline is not a mandatory step for indexing as Elastic Agent does seamless integration. One of the examples of ingest pipelines written for the logs from test builds is shown in Figure 13. An example of the Ingest pipeline shown in Figure 13 shows that "Grok" and "Remove" processors are stacked in series.



```
[
  {
    "grok": {
      "field": "message",
      "ignore_missing": true,
      "patterns": [
        "%{GREEDYDATA:junk_number} %{TIMESTAMP_ISO8601:alert_time} %{WORD:host_name}
%{WORD:application_process_name} %{NUMBER:process_id} - -  %{GREEDYDATA:log_text}"
      ],
      "description": "Extract fields from 'message'"
    }
  },
  {
    "remove": {
      "field": "message"
    }
  }
]
```

Figure 13: Ingest Pipeline for logs generated from test builds which are to be indexed into Elasticsearch

Log data used in this research comes from different platforms and does not follow a uniform format for all generated logs from multiple sources. Thus we need a bridge in between to flatten the incoming data to a common ground. Here, Elastic Common Schema comes to the rescue to ensure that all the incoming data are "speaking the same language". This helps to make common field names for the incoming data from multiple sources. This approach made the process of query using KQL a lot easier as it was possible to run queries in a central logging station. The Elasticsearch data stream concept is used in the research as log files have timestamps in each log line and thus log lines can be interpreted as append-only time series data. Data streams are used by Elastic Agent for all available integrations. After ingesting data using the ingest pipeline to create a data stream into Elasticsearch, the Kibana Discover window is segmented into different custom fields as shown in Figure 14.



Pri...	Facility	alert_time	hostname	Application-name	Process-id	log_message	log.file.path
1	<135>	2022-11-16T09:44:02.764207+00:00	44-4b-5d-00-00-49	axone_comm	627	0|main|98|IpcReceiver.hpp:46|ppm::ipc::IpcClient - Receive needed callback	/home/ubuntu/Desktop/datasets/version5/syslog134
1	<134>	2022-11-18T10:52:28.027474+00:00	qemux86	data	975	0|main|56|AlgorithmTrace.cpp:29|/home/abuild/rpmbuild/BUILD/wearable-algorithms-014.31.f.176/Develop/EkproV12/Code/EkPro.cpp:197:Relearn initiated	/home/ubuntu/Desktop/datasets/version5/syslog17
1	<135>	2022-11-16T09:44:02.357435+00:00	44-4b-5d-00-00-49	axone_comm	628	0|main|72|Persisted.hpp:333|Loading SHA 256 checksum file: /home/axone_data/persistedPatientAssignment.data.checksum	/home/ubuntu/Desktop/datasets/version5/syslog99
1	<135>	2022-11-16T09:44:02.188206+00:00	44-4b-5d-00-00-49	data	615	0|main|181|NetworkCommandSender.cpp:69|CommitSetServersResponse, status:0	/home/ubuntu/Desktop/datasets/version5/syslog106
1	<135>	2022-11-18T11:30:12.604216+00:00	qemux86	simulator	769	0|3068947264|16|SensorAssociationSimulator.cpp:1267|Send D1-packet to outbound interface 1 with versionAndType: 22, port: 40, source address: 2, destination address: e1, msgId: 76	/home/ubuntu/Desktop/datasets/version5/syslog31
1	<30>	2022-11-16T09:43:50.187735+00:00	44-4b-5d-00-00-49	systemd-networkd	628	usb1: Gained IPv6LL	/home/ubuntu/Desktop/datasets/version5/syslog76
1	<28>	2022-11-16T09:43:50.268351+00:00	44-4b-5d-00-00-49	systemd-networkd	629	wlan0: found matching network '/etc/systemd/network/10-wireless.network', based on potentially unpredictable interface name.	/home/ubuntu/Desktop/datasets/version5/syslog45
1	<135>	2022-11-20T19:13:21.707804+00:00	qemux86	data	662	0|main|300|ProtocolStateMachineAgent.cpp:807|45323#011<A0A> Gen:Debug: SapCommon_StateMachine_StateChanged (MeasurementsEnabled->ActivateMeasurements)	/home/ubuntu/Desktop/datasets/version5/syslog162
1	<134>	2022-11-16T09:44:02.035127+00:00	44-4b-5d-00-00-49	axone_comm	625	1668591842.028 (5) axone_comm[625/1968842672] <L> 0|main|71|AxoneNetEndpoint.cpp:381|Use Real patient registry	/home/ubuntu/Desktop/datasets/version5/syslog125
1	<135>	2022-11-18T10:08:23.903637+00:00	qemux86	data	754	0|main|253|ProtocolStateMachineAgent.cpp:807|0#011<A09> Gen:Debug: SapCommon_Dispatcher_ProcessedDPacket (Output:vt=23,s=e1,d=2,pm=4076)	/home/ubuntu/Desktop/datasets/version5/syslog20
1	<135>	2022-11-16T09:43:54.087668+00:00	44-4b-5d-00-00-49	hub_gui	634	0|main|74|UiConnectionManager.cpp:107|HW IpcClient - no msgs received	/home/ubuntu/Desktop/datasets/version5/syslog108
1	<135>	2022-11-18T11:31:30.577759+00:00	qemux86	axone_comm	760	0|main|240|StreamWriter.hpp:186|Creating umfwav_writer publisher, topic umfwav, streamId 47899162-f72b-4ebe-9bb8-9ab4b1e814e4	/home/ubuntu/Desktop/datasets/version5/syslog14
1	<31>	2022-11-16T09:43:50.731934+00:00	44-4b-5d-00-00-49	Middlewar	589	daemon: NTP active already	/home/ubuntu/Desktop/datasets/version5/syslog121
1	<134>	2022-11-16T09:43:53.645010+00:00	44-4b-5d-00-00-49	hub_gui	641	0|main|68|UiSettingsManager.cpp:119|New hub udi: (01)00195278093998(21)SRW20440015SP(11)201102	/home/ubuntu/Desktop/datasets/version5/syslog90
1	<135>	2022-11-20T19:13:09.362691+00:00	qemux86	data	664	0|main|263|ProtocolStateMachineAgent.cpp:807|46310#011<A0A> Gen:Debug: SapHub_Session_Out_CapabilityQueryReq (0,e1,3)	/home/ubuntu/Desktop/datasets/version5/syslog161
1	<135>	2022-11-16T09:43:49.646721+00:00	44-4b-5d-00-00-49	axone_comm	625	0|main|51|Persisted.hpp:427|Calculated checksum 4e7fbd2e2713f9f398c23ef05deda16f9cefeeed0f6b057aa926a263544ea8d0 for /home/axone_data/persistedPatientAssignment.data	/home/ubuntu/Desktop/datasets/version5/syslog120

Figure 14: Kibana Discover window after data is ingested using ingest pipeline

2.4 Elastic Stack as an Exploratory Data Analysis Tool

After the ingestion of logs into Elasticsearch using custom log integration and Elastic Agent, a centralized logging system was established. This enabled us to perform complex queries and keyword searches from logs acquired from various sources. KQL was used to perform queries. KQL is a simple and efficient text-based language designed specifically for data filtering purposes. KQL, however, is limited to data filtering and does not provide features for data aggregation, transformation, or sorting. A comprehensive overview of Kibana features can be found in the official documentation of Elastic [16].

Upon execution of certain queries, immediate access to their distribution data was possible. This enabled rapid identification of respective sources of log data along with the distribution. The executed queries provided the number of hits per query and identification of the corresponding log lines. The Elastic Stack also offered the advantage of exploring surrounding documents via a user-friendly interface. This made the navigation and exploration of logs in a large volume convenient and efficient.

The integration of Elasticsearch with Kibana allowed for in-depth visualization and understanding of distribution parameters such as hostname, application name, file paths, and process IDs. This was a valuable tool for constructing datasets, enabling the filtering out of numerous log files that lacked significant features. As a result, we could focus only on log files as well as their sources that had interesting errors and anomalies. This drastically improved efficiency and data quality prior to the application of machine learning algorithms. The careful preprocessing steps are essential to ensure that machine learning models are trained on meaningful and relevant data, enhancing their performance and predictive power. Figure 15 shows the distribution of the 'application-name' field in the log message.

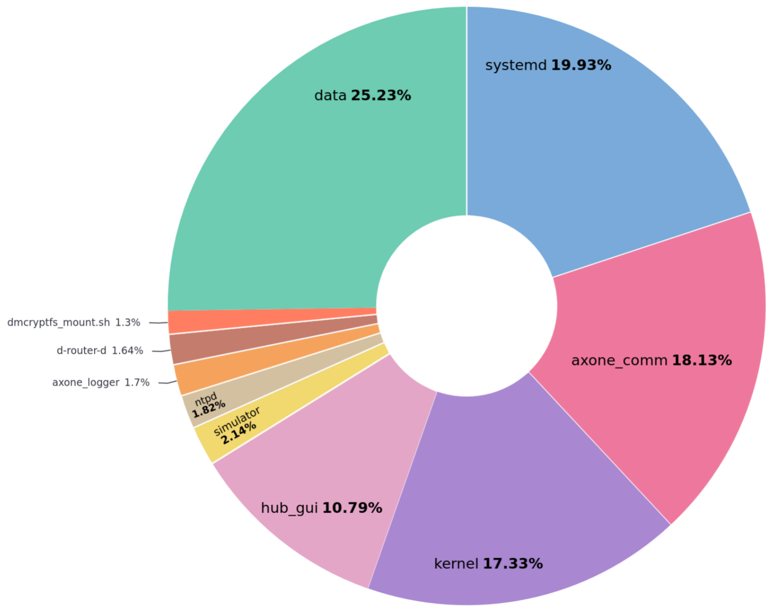


Figure 15: Pie-chart showing the distribution of application name along with the path of their source files

3 Logs as Text Embeddings Spaces

In mathematics, embedding roughly refers to a mapping between different representation spaces in a way that specific properties are preserved [17]. It can be inferred from the definition that a word embedding is some kind of representation bounded by some function that is mapped from one space to another that has similar properties. Word embedding is represented by a d-dimensional vector $w \in \mathbb{R}^d$.

3.1 Tokenization

Almost all of the text processing tasks either traditional or recent state-of-the-art begin with tokenization. Tokenization is a method of splitting or transforming a text, such as a sentence or document into sub-components called tokens. These tokens are stored in a list and these collections of all the tokens produced from a text corpus are represented as vocabulary, often denoted as $\mathcal{V}$. The size of vocabulary $|\mathcal{V}|$ has a direct influence on almost all language modeling tasks. Mainly tokenization is achieved by splitting the text into words, subwords, or characters. The method of splitting text into words or subwords differs from the morphological structure of the languages. Since the logs used in this research are based in the English language, the tokenization procedure used in this literature is based on English text.

The process of tokenization of text into tokens of word, subword, or characters is achieved by the use of tokenizers. Generally, the tokenizers used in language processing can be categorized into rule-based and trainable tokenizers. Rule-based tokenizers mostly follow the technicality of linguistics to split text into words or characters. They mostly leverage the characters to split text into words like whitespace, commas, and punctuation to split the text into smaller units. There are several libraries developed for achieving this tokenization some of them are Spacy [18], which uses whitespaces and punctuation to split words. They have separate hard-coded rules for cases like $"haven't" \Rightarrow ["have", "n't"]$.

This rule-based method of tokenization is an effective technique but it comes with certain limitations. As log messages are written for the ease of developers, they aren't formed with clean sentence formation obeying the rules of linguistics. For example, in the example log message shown in Figure 16, splitting the log message into smaller units by whitespace and punctuation doesn't lead to consistent tokens for the same text. For example, there are single terms like *"dateOfBirthRequired"*, *"lastNameRequired"*, *"pid1NomenclatureDesignation"*, *"sampleInfo"*, $"nulllast\_sample? = tr"$ which are composed of combination of different words. Similarly, there is extensive use of special characters and they are used inconsistently which makes it difficult to operate the use of the special characters in splitting text to produce uniform tokens for the similar text. This causes the problem of out-of-vocabulary (OOV) words. These OOV words are those words or tokens that are not present or filtered out in training vocabulary but appear during testing or inference/prediction state. For example, if there are words like required, admission, and fields separately are present in a training vocabulary but the prediction log comes up with a word *"requiredAdmissionFields"*, this word is treated OOV word. There are numerous cases of these characteristics in the log data used in this research as log messages don't abide by common and consistent standards. The common approach to dealing with OOV words in language modeling is the use of special tokens such as $[UNK]$ or pad with zeros or unique random representation of each OOV word [19].

```
"2023-04-11T11:45:36.893+0000","87274","87274","INFO","um-pm-0.
    novalocal","6bca2187-ec97-4f89-bac7-98ac691464d5","axone-config","
    axone-configMain[30934:112]","com.ge.hc.lcs.axone.configservice.
    message.ConfigMessageProcessor:331","General","RECEIVED:
    Subscription - ConfigStateMessage{configState=: umf_cfg_state :
    cfgContextHash: f5a3ae6fc9759da4c7f93fee1c6b2af0011f218a
    cfgContext: 0: name: facility value: Cleveland Clinic Main Campus
    1: name: factory value: GE scope: //params/system/cfg/policy/
    pid1NomenclatureDesignation, //params/system/cfg/policy/
    pid2NomenclatureDesignation, //params/system/cfg/policy/
    requiredAdmissionFields/dateOfBirthRequired, //params/system/cfg/
    policy/requiredAdmissionFields/firstNameRequired, //params/system/
    cfg/policy/requiredAdmissionFields/genderRequired, //params/system
    /cfg/policy/requiredAdmissionFields/lastNameRequired, //params/
    system/cfg/policy/requiredAdmissionFields/pid1Required, //params/
    system/cfg/policy/requiredAdmissionFields/pid2Required ,info=
    SampleInfo:[ sample_state=NOT_READ_SAMPLE_STATE view_state=
    NOT_NEW_VIEW_STATE instance_state=ALIVE_INSTANCE_STATE
    instance_handle=2f287325-c370-2345-dd0c-f007d29b94af valid_data=
    true sample_rank=0 generation_rank=0 absolute_generation_rank=0
    reception_timestamp=[sec=1681213536,nanosec=892564998]
    publication_sequence_number=[high=0,low=2363]
    reception_sequence_number=[high=0,low=15777]
    original_publication_virtual_guid=a8350a9f-cdd3-c596-864c-0
    f5480000002 original_publication_virtual_sequence_number=[high=0,
    low=3] source_guid=a815ff19-bc59-1b8c-d9da-88d680004202
    topic_query_guid=null last_sample?=tr
```

Figure 16: Syslog generated from medical devices in hospital settings

To mitigate the problem of OOV words in language modeling, a different tokenization method of splitting words into sub-words was introduced. This kind of tokenization method is referred to as trainable tokenization. These methods such Byte-Pair-Encoding [20], WordPiece [21], Sentence-Piece [22], take a statistical approach to disintegrate a word into smaller sub-units. During the training process, the first analysis is run on a text corpus, and unigram probabilities are computed for substring occurrences. This facilitates the tokenizer to learn the distribution of substrings and thus it can define the optimal way to split the text and create a vocabulary of words. This method of tokenization helps to counter the problem of OOV words that could potentially arise at the prediction time of log lines. For example, the word *"Filemanager"* is tokenized into ["File", "man", "nag" "age", "ger"]. Thus these sub-units are fit for words like File, Manager, and some extensions as well file *filemanager.txt*.

3.2 Word Embeddings Algorithms

Word embeddings are used in Natural Language Processing to convert words into numerical vectors. The process of converting words into vectors is known as text vectorization. One of the simplest word vectorization techniques is One-Hot Encoding. One-hot encoding creates a vector for each word in a vocabulary, where all the vector values are labeled as 0 except the index of the word which is labeled as 1. One-Hot Encoding creates word vectors in multiple dimensions. The number of dimensions is equal to the total number of words in the vocabulary.

Word embeddings are a powerful natural language processing (NLP) technique that converts words into fixed-size numerical vectors, capturing semantic and syntactic relationships between words. Word embeddings have been successfully applied in various NLP tasks, such as sentiment analysis, machine translation, and text classification. This paper discusses the potential of word embeddings for system log analysis, highlighting their benefits and possible applications. Word embeddings represent words in a high-dimensional vector space, where semantically similar words are placed close together [23]. This representation enables the application of mathematical operations on words, capturing relationships like analogy, synonymy, and autonomy. The two most popular methods for generating word embeddings are Word2Vec [23] and GloVe [24]. Both techniques rely on unsupervised learning from large text corpora, leveraging co-occurrence statistics to generate continuous word vectors.

3.2.1 Static Word Embeddings

A static word embedding model is a function that maps each word type to a single vector. These vectors are typically dense and have much lower dimensions than the size of the vocabulary. This is a significant advantage to one-hot-encoding as one-hot encoding has a dimension the same as the size of the vocabulary and this creates serious problems when applied to a text corpus with a large vocabulary size. There have been multiple approaches to developing this mapping function to create static word embeddings. This mapping function assumes the fixed size vocabulary so rare words are treated as *UNK* tokens such that OOV words during prediction are also treated as *UNK*. Neural networks based models were used by Mikolov et al. to develop Word2Vec [23] and FastText [25], matrix factorization was used to create word embeddings in GLoVE [24]. Other approaches like probabilistic methods based on abstract mathematical concepts have also been explored in the realm of static word embedding techniques but however, but in this research, we primarily focus on the Word2Vec family of algorithms for the generation of static embeddings.

Word2Vec Algorithm

Word2Vec is a two-model architecture developed by Mikolov et al. [23] for computing continuous vector representation of words from large data sets. It consists of a shallow neural network with an input layer, one hidden layer, and an output layer with a softmax activation function. The training of this neural network-based architecture is based on two different context representations: continuous bag-of-words (CBOW) or skip-gram model. The high-level overview of the context representation is shown in Figure 17.

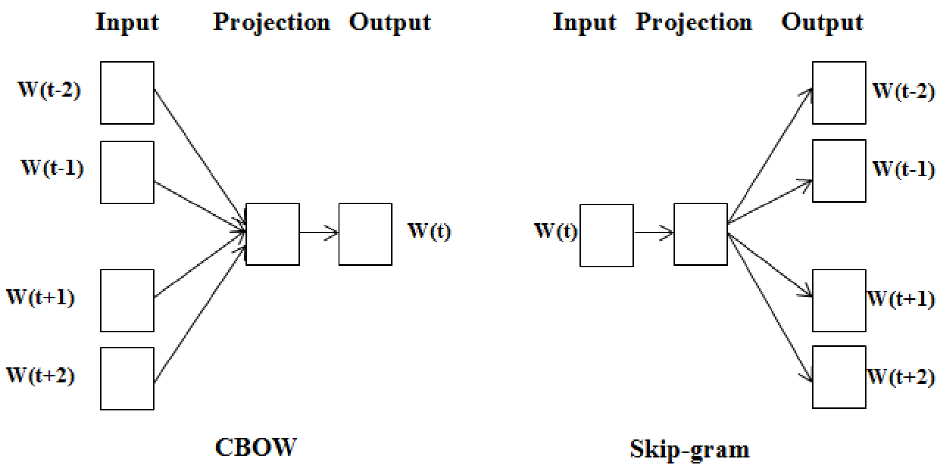


Figure 17: Word2Vec model architectures. The CBOW architecture predicts the current word based on the context, and the skip-gram architecture predicts surrounding words given the current word [23].

The inputs for the Word2Vec algorithm are pairs *(w, C)*, where w is a single word and *C* is the context surrounding *w*. These inputs are fed to the input layer of the neural network as one-hot-encoded vectors of size 1 x $|\mathcal{V}|$. The output of the algorithm is a vector of probabilities of *z* of all words in the vocabulary $\mathcal{V}$ [17].

The difference between CBOW and skip-gram methods is for CBOW, the context C is taken as input and a single word w is predicted. Whereas in the skip-gram method, the process is exactly the opposite as a single word is given as input, and the context C is the target word to predict. Word order in a text is ignored for both of the methods meaning that Word2Vec does not preserve the positioning of text in a sentence. Similarly, two different optimization objectives are used - negative sampling or hierarchical softmax. Both of the models of Word2Vec namely, skip-gram and CBOW are relevant to this thesis work. Thus, this section walks through the introduction and training procedures of Skip-Gram and CBOW respectively. The high-level architecture of the two model families of Word2Vec according to the original paper by Mikolov et al. [23] is given in Figure 17.

Skip-Gram Model

The skip-gram model assumes that the word can be used to generate its surrounding words in a text sequence. For example, the text sequence in the log line - The sensor module configuration has changed. Let's choose *"configuration"* as the center word and set the context window size to 2. As shown in Figure 18, given the center word "configuration", the skip-gram model considers the conditional probability for generating context words: "The", "sensor", "module", "has", "changed", which are no more than two words away from the center word can be written as

$$P(\text{"}sensor\text{"}, \text{"}module\text{"}, \text{"}has\text{"}, \text{"}changed\text{"} \mid \text{"}configuration\text{"}) \tag{1}$$

The skip-gram model assumes that the context words are independently generated given the center words. Thus, the above conditional probability can be written as,

$$\begin{aligned} &P(\text{"}sensor\text{"} \mid \text{"}configuration\text{"}).P(\text{"}module\text{"} \mid \text{"}configuration\text{"}).\\ &P(\text{"}has\text{"} \mid \text{"}configuration\text{"}).P(\text{"}changed\text{"} \mid \text{"}configuration\text{"}) \end{aligned} \tag{2}$$

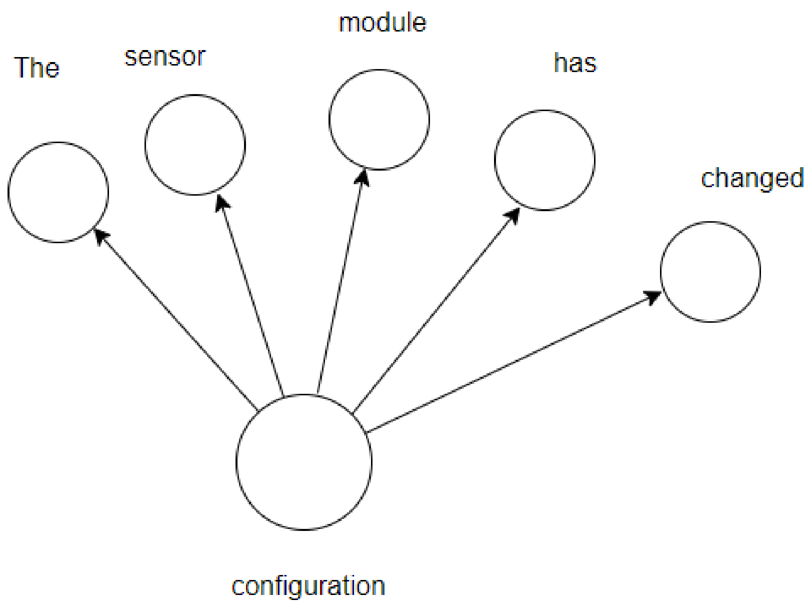


Figure 18: Conditional probability of generating the surrounding context words given a center word.

In the skip-gram architecture, each word is represented by two vectors, each of dimension d, for the purpose of determining conditional probabilities. Specifically, for a word indexed as i within the vocabulary, it has two vector representations, $\mathbf{v}_i \in \mathbb{R}^d$ and $\mathbf{u}_i \in \mathbb{R}^d$ when used as a *center* word and a *context* word, respectively. Given the center word w_c (with index c in the dictionary), the conditional probability of generating any context word w_o (with index o in the dictionary) can be modeled by a softmax operation on vector dot products as shown in Equation 3.

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{3}$$

In the Equation 3, the vocabulary index set is represented as $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$. Given a text sequence of length T, where the word at time step t is denoted as $w^{(t)}$. It is assumed that context words are independently generated given any center target word. The likelihood function of the skip-gram model for context window size m is the probability of generating all context words given any center word as,

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w^{(t+j)} \mid w^{(t)}), \tag{4}$$

where any time step that is less than 1 or greater than T can be omitted.

The skip-gram model parameters are the center word vector and context word vector for each word in the vocabulary. In training, the aim is to maximize the likelihood function (i.e., maximum likelihood estimation) to learn model parameters optimally. This is equivalent to minimizing the following loss function in equation 5.

$$-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{5}$$

When using stochastic gradient descent for loss minimization, shorter sub-sequences are randomly sampled in each iteration. This sampling aids in determining the gradients for that specific sub-sequence, which is then employed to update the model's parameters. To compute the gradients, it's imperative to derive the gradients of the log conditional probability relative to both the center word vector and the context word vector [26]. In general, according to equation 3 the log conditional probability involving any pair of the center word w_c and the context word w_o is,

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right). \tag{6}$$

Through differentiation, we can obtain its gradients with respect to the center word vector $\mathbf{v}_c$ as

$$\begin{aligned} \frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c)\mathbf{u}_j. \end{aligned} \tag{7}$$

The calculation in 7 requires the conditional probabilities of all words in the dictionary with w_c as the center word. The gradients for the other word vectors can be obtained in the same way.

After training, for any word with index i in the dictionary, we obtain both word vectors $\mathbf{v}_i$ (as the center word) and $\mathbf{u}_i$ (as the context word). In natural language processing applications, the center word vectors of the skip-gram model are typically used as the word representations. [26]

The Continuous Bag Of Words (CBOW) Model

The continuous bag of words (CBOW) model assumes that a center word is generated based on its surrounding context words in a text sequence. Let’s take the same example as in the skip-gram model, "The sensor module configuration has changed " where *"configuration"* is the center word. The context window size is 2 as in the skip-gram example, continuous probability of generating center word *"configuration"* based on the context words "sensor", "module", "has", and "changed" is

$$P(\text{"}configuration\text{"} | \text{"}sensor\text{"}, \text{"}module\text{"}, \text{"}has\text{"}, \text{"}changed\text{"}) \tag{8}$$

The word *"The"* has been omitted because it is the third word from the center word and the context window size is 2. This representation is shown in Figure 19.

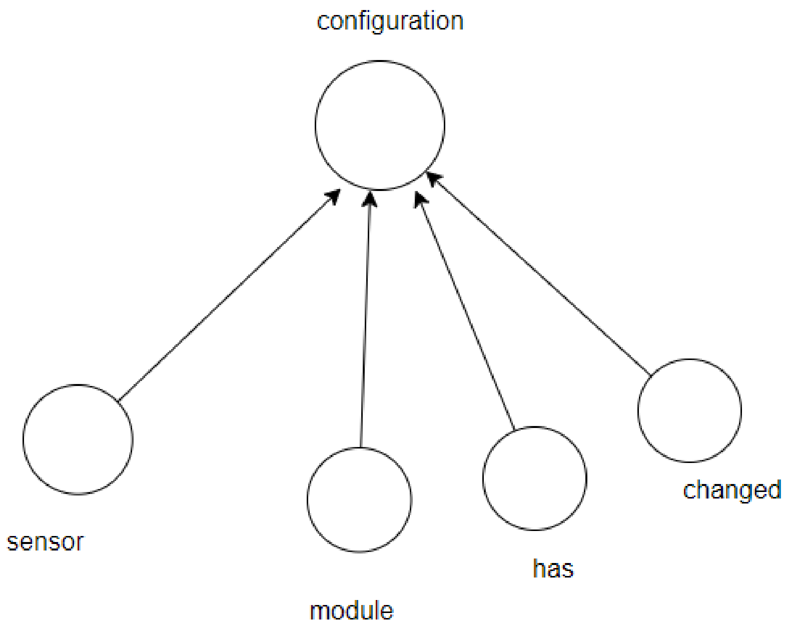


Figure 19: Conditional probability of generating the center word given surrounding context words.

The choice of a number of context words is a hyperparameter that can be tuned during the training process. These context word vectors are averaged in the calculation of the conditional probability. The conditional probability of generating any center word w_c (with index c in the dictionary) given its surrounding context words $w_{o_1}, \ldots, w_{o_{2m}}$ (with index $o_1, \ldots, o_{2m}$ in the dictionary) can be modeled by the Equation 9.

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)} \tag{9}$$

which can be simplified as,

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i\in\mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{10}$$

Given a text sequence of length T, where the word at time step t is denoted as $w^{(t)}$. For a context window of size m, the likelihood function of the CBOW model is the probability of generating all center words given their context words, which can be represented as,

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{11}$$

The training process of the CBOW model is almost similar to training skip-gram models. The maximum likelihood estimation of the CBOW model is equivalent to minimizing the following loss function:

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{12}$$

It can be noticed that,

$$\log\ P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log\left(\sum_{i\in\mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)\right). \tag{13}$$

We can obtain its gradient with respect to any context word vector $\mathbf{v}_{o_i} (i = 1, \ldots, 2m)$ through differential equation as,

$$\begin{aligned}\frac{\partial \log\ P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} &= \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j\in\mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{\sum_{i\in\mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}\right) \\ &= \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j\in\mathcal{V}} P(w_j \mid \mathcal{W}_o)\mathbf{u}_j\right).\end{aligned} \tag{14}$$

The gradients for the other word vectors can be obtained in the same way. Unlike the skip-gram model, the continuous bag of words model typically uses context word vectors as the word representations.

3.2.2 Contextual Embeddings Models

Contextual embedding models as the name suggests are those models that take into account the context in which a word appears in a text to generate word representations of a text document. Unlike static word embedding models like Word2Vec, Glove, etc. which assign a fixed representation of word token, the contextual embedding models generate different vector representations for a word depending on the context where it appears. For a token x, contextual embedding can be represented by a function $f(x, c(x))$ depending on both x and its context $c(x)$. With the empirical success of static embeddings, contextual embeddings attracted a lot of research communities considering the importance of context in text analysis. Some seminal works in context embeddings include TagLM(Semi-supervised sequence tagging with bidirectional language models) [27], ELMo(Embeddings from Language Models) [28].

All the details of ELMo are out of the scope of this thesis so the interested readers can look at the paper [28]. Although improved solutions compared to static embeddings model for language understanding it was task-specific architecture. Keeping this in mind, the task agnostic model was introduced in the form of GPT (Generative Pretrained Transformer) [29] architecture to generate contextual embeddings. It encodes context from left to right in a unidirectional format. It was built on a Transformer Decoder and it is a pre-trained language model representing text sequences. The Decoder of the Transformer is explained in Section 4.2.

As unidirectional language encoder GPT turned out to be a huge success, the directional nature of encoding for task agnostic tasks was introduced in the form of BERT (Bidirectional Encoder Representation from Transformers) [30]. BERT encodes the context words in both directions left and right in a text sequence, and requires minimal changes for a broad range of natural language processing tasks. BERT acts as an excellent choice to extract features from a text sequence and it has been used as a feature extractor in many diverse tasks. This research also experiments with BERT as a feature extractor for the log sequences.

Words in text don't appear in isolation and this holds true in log lines. The use of the special key word means nothing if the context is not known. There are many cases of words and collections of words in log lines in a file which might represent anomaly or normal depending on the context within a log line or prior log events. Also, the different texts might mean the same thing in the log line because of the inconsistent use of whitespaces and special characters. Thus it is extremely important to learn the representations of each word in log lines. BERT is useful to capture nuances in word meanings in log lines that can change depending on the typos or inconsistent use of white spaces and characters.

BERT as Contextual Embedding Model

Bidirectional Encoder Representations from Transformers, known commonly as BERT is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers [30]. As the name suggests it is based on an encoder of transformer architecture [31] and applies a transformer encoder to attend bidirectional contexts during pretraining. It proposes masked language modeling and next-sentence prediction (NSP). In masked language modeling, some of the tokens of an input sequence are randomly masked and the objective is to predict these masked positions taking the corrupted sequence as input. In addition, BERT uses NSP where the given two sentences, NSP predicts whether the second sentence is actual/true next sentence of the first sentence. BERT uses special tokens to obtain a single contiguous sequence for each input sequence. The first token is [CLS] which is a classification token and sentence pairs are separated using [SEP] token. The final hidden state of [CLS] token is used for sentence-level tasks and the final hidden state of each token is used for token-level tasks.

In particular, the final hidden state corresponding to the [CLS] token is used as the aggregate sequence representation for sentence-level tasks. Alternatively, the final hidden state for each token can be used for token-level tasks. This is represented by the following equation:

$$C = BERT_{CLS}(x_1, x_2, ..., x_n). \tag{15}$$

where C is the output vector corresponding to the [CLS] token and x_i represents the input tokens.

However, utilizing the [CLS] token for sentence embedding is not a strict requirement. The BERT framework allows flexibility in the manner one can obtain the final representation of the sequence. A popular alternative to using [CLS] token is to perform a pooling operation to the final hidden states of all tokens in the sentence. Various pooling mechanisms such as mean pooling and median pooling can be employed.

Mean pooling calculates the average of the final hidden states of all tokens. Mathematically, it is represented by:

$$C_{mean} = \frac{1}{n}\sum_{i=1}^{n} BERT(x_i). \tag{16}$$

On the other hand, median pooling selects the median value among the final hidden states. These alternative methods provide different benefits. Mean pooling takes into account the semantic information from all tokens in a sentence and thus, can capture more context. However, it may be influenced by outlier values. Median pooling is less susceptible to outliers, offering a more robust representation.

The choice of the pooling method depends on the specific application. In the context of log line feature extraction, the pooling mechanism should be chosen considering the nature of log data. If the logs contain outliers or high variance in their semantic content, median pooling may be a better choice. On the other hand, if the logs are relatively uniform and all tokens contribute equally to the overall semantics, mean pooling can provide a better representation. Thus, BERT, with its flexible mechanisms, offers powerful options for log line feature extraction.

4 Machine Learning and Natural Language Processing

Machine learning (ML) enables computers to extract knowledge from data, enhance performance through experience, and make forecasts on unseen examples. In the case of supervised machine learning, it leverages labeled data for learning, i.e, each piece of training data is associated with a specific label. The objective of this learning approach is to identify the hidden pattern in the data, enabling accurate predictions on unseen data exhibiting a similar pattern. This method has found significant applications in log analysis, particularly in the identification of anomalies in log files. Such anomalies could stem from equipment malfunction or a cyber intrusion in the network. The ability to pinpoint these deviations helps in mitigating fraudulent activities, enemy attacks, and network breaches that could jeopardize the company's future. In the context of log analysis with supervised learning, the dataset used for anomaly detection consists of both normal and anomalous examples, each accurately labeled. However, log files, while textual in nature, may not consistently exhibit a structured format. They can often be semi-structured or unstructured, with a multitude of special characters and tokens. Unlike natural languages, they lack morphological richness and do not offer much context. They usually consist of a sequence of lines generated in a particular order as seen in Section 2.1. The progression in natural language processing algorithms, such as BERT and Transformer, enables the conversion of semantic information from log files into mathematical vector representations. This capability allows us to uncover hidden patterns that would otherwise be indiscernible without precise feature engineering.

The following subsections discuss different existing machine learning architectures applied in the area of NLP for identifying anomalies within log files. Section 4.1 presents the theoretical details of recurrent neural network architectures practiced in the area of NLP. Section 4.2 provides a description of the transformer architecture, recognized as a seminal work in the domain of NLP.

4.1 Recurrent Neural Networks (RNNs)

Log files represent data as sequences of events occurring over time. This sequential nature makes it crucial to consider the temporal dependencies between different events in the sequence. RNNs are particularly adept at handling such temporal sequences. They achieve this by allowing previous outputs to be reused as inputs in subsequent steps, maintaining hidden states that effectively capture the memory of past events. As depicted in Figure 20, this unique architecture of RNNs enables them to model long-term dependencies in data, making them highly suitable for tasks where context from earlier steps is essential for accurate predictions. Their ability to process and remember sequential data has led to their successful application in various domains, including natural language processing and time-series forecasting. When applied to log files, RNNs have shown promising results, particularly in tasks like anomaly detection. The study by [32] is a testament to their efficacy, demonstrating how RNNs can be employed to identify unusual patterns or outliers in log sequences, helping in proactive system monitoring and timely detection of anomalies in log data.

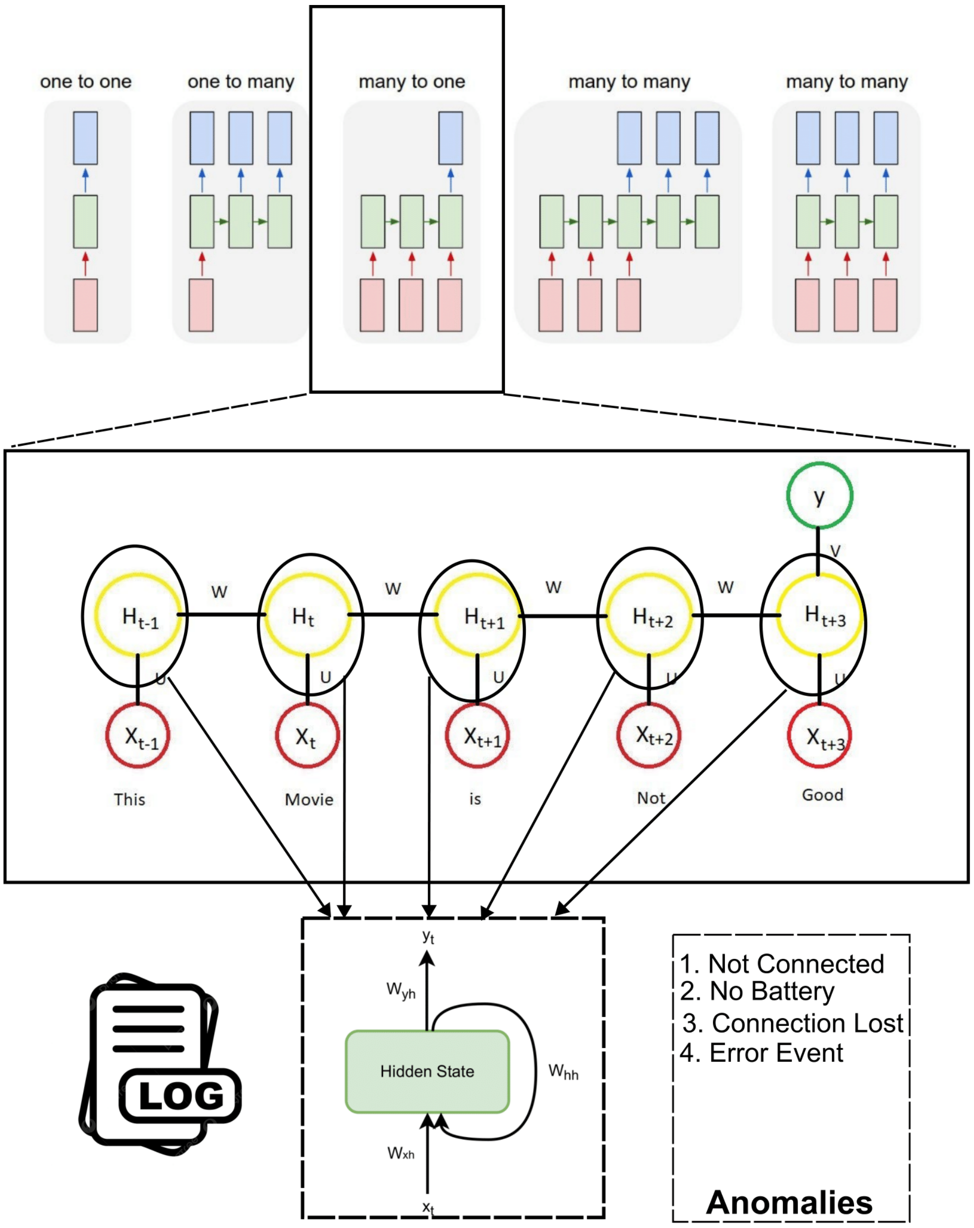


Figure 20: Diagram showing RNNs capturing sequential data using hidden states.

There are various versions of RNNs, such as many-to-many (employed in language translation) and many-to-one (utilized in anomaly detection), among others. When it comes to identifying and classifying anomalies from log files, the process begins by transforming log texts into vector format using word embeddings. Following this, there are multiple hidden layers, each of which applies an affine transformation function (like a matrix dot product multiplication), succeeded by a non-linear function. The final hidden layer is followed by an activation function, resulting in an output layer. This output layer predicts or classifies the input into various types of anomalies. However, RNNs suffer from vanishing gradient problems. The vanishing gradient problem is characterized by the significant reduction of gradient values during backpropagation over time, which can result in slow learning or complete stagnation of learning in the initial layers of the network. To overcome these problems, Long Short-Term Memory (LSTM) networks have been proposed.

LSTM cell in an LSTM network uses a memory block that makes it convenient for a network to store a given input over many time steps [33]. LSTM is able to capture temporal dependencies through connections between LSTM units in sequential data that allow information to cycle through a loop across adjacent time steps. It creates an internal state of feedback, allowing the network to understand the temporal information present within the data. The memory state in LSTM allows the network to selectively forget and remember information. This allows information of higher importance to be retained and backpropagate and exclude irrelevant information simply by forgetting using the forget gates. Each block in LSTM has three gates: the input gate, the forget gate, and the output gate. Gates allow information to pass through selectively using sigmoid activation function:

$$\sigma(t) = \frac{1}{1 + e^{-t}}. \tag{17}$$

This passes information when true, else no information is passed into the gate. The equation of gates is:

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i), \tag{18}$$

$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f), \tag{19}$$

$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o). \tag{20}$$

where i_t,f_t and o_t are input gate, forget gate, and output gate respectively. w_j and b_j refer to weight and bias of the respective gate (j) and σ refers to sigmoid activation. Further, output vector h_t and cell state vector c_t is calculated as:

$$\tilde{c_t} = tanh(w_c[h_{t-1}, x_t] + b_c), \tag{21}$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}, \tag{22}$$

$$h_t = o_t * tanh(c_t). \tag{23}$$

Where $\tilde{c_t}$ represents the vector on how much update each state vector requires, tanh is hyperbolic tangent activation. A single unit of LSTM block is shown in Figure 21.

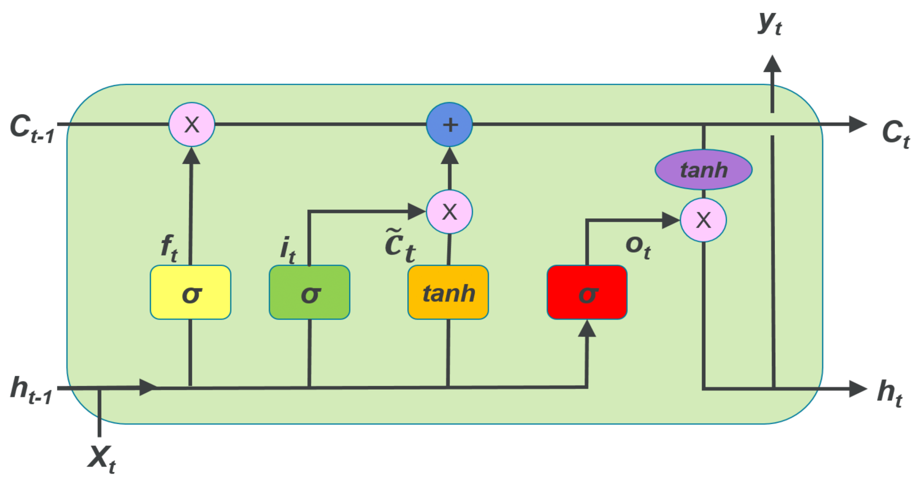


Figure 21: A block diagram of a single unit in an LSTM network

4.2 Transformers

Introduced in 2017 [34], Transformers were designed to address sequence-to-sequence tasks, excelling in managing long-term dependencies. This offers a significant advantage over traditional RNNs and LSTM networks that struggle with issues such as vanishing gradients or computational inefficiency when dealing with long sequences. An important innovation in Transformers is the implementation of the self-attention mechanism, which allows the model to weigh the importance of tokens within the sequence dynamically.

A Transformer model is composed of an encoder-decoder architecture. The encoder processes the complete input sequence, condensing it into a fixed-length vector representation. This encoded information is then passed to the decoder, which generates the output sequence from the fixed-length vector. Transformers incorporate the sequence order of words in the input by utilizing positional encoding. In this method, each word embedded in the input sequence is embedded with a corresponding positional encoding vector, which represents its location in the sequence.

The operation of a Transformer generally follows these steps [35]:

- The input sequence is transformed into word embeddings and then, positional encoding encodes the position information of each word, which is fed into the first encoder.
- The output from the first encoder is further transformed and propagated through the subsequent encoders in the stack.
- The final encoder's output in the stack is forwarded to all the decoders in the decoder stack, setting the stage for generating the output sequence.

This sequence of operations allows Transformers to handle tasks that require understanding the context and dependencies in an input sequence, such as machine translation, text summarization, and more.

Attention mechanism in Transformer: Rather than relying solely on preceding hidden vectors within the input word embedding, the attention layer in a network strategically focuses on a weighted average of all word embeddings. By doing so, it can capture contextual relationships between words regardless of their positions in the sequence, ensuring that the semantic essence of the text is maintained.

In the Transformer architecture, this attention mechanism is executed by transforming word embeddings into three distinct matrices: queries, keys, and values. Each word in the sequence gets its own query, key, and value representation through this transformation, which is achieved by multiplying the word embeddings with three separate weight matrices that are learned during training. The relationship between these matrices is fundamental to the attention mechanism. The query from one word interacts with the key of every other word to generate an attention score, determining how much focus the model should place on that word.

To illustrate, consider the sentence "False input leads to error". In trying to represent the significance of the word "False", the attention layer calculates these attention scores by comparing the query of "False" with the keys of all words in the sentence. The resulting scores dictate the importance of each word in relation to "False". After obtaining these scores, they are used to take a weighted average of the value representations of all words, producing a contextually rich representation for the word "False". This dynamic allocation of focus allows the model to better understand the intricate relationships between words in the sequence, especially when faced with longer sentences or more complex semantic structures.

Furthermore, the self-attention mechanism's ability to provide different weights for different words gives the Transformer model its unique capability to handle a variety of tasks, from machine translation to sentiment analysis. Over time, as the model is trained, these weights (residing in the query, key, and value matrices) are continuously updated, ensuring that the model's representation of sequences becomes more accurate and nuanced. This continual refinement allows Transformers to capture subtleties in language that many traditional models might overlook.

4.3 Machine Learning Deployment and Optimization

4.3.1 Large Language Model (LLM) Compression

Even though large language models have taken over NLP with their performance, they are computation-hungry and demand a lot of resources for training, inference, and subsequent deployment. The weights and parameters of these large models are in 10s to 100s of Gigabytes making it challenging to deploy in low-resource applications. Thus, it has gathered huge research interest to compress the models to reduce model footprints and enable faster computation because of the high performance of these large transformer models. The compression must be able to reduce the model's size without significantly hurting the performance. The four approaches of compressing the large models practiced in research and industrial community presented in a survey paper by Cheng et al. in 2020 [36] are as follows:

- Parameter Pruning and Quantization
- Low-rank Factorization
- Transferred/Compact Convolution Filters
- Knowledge Distillation

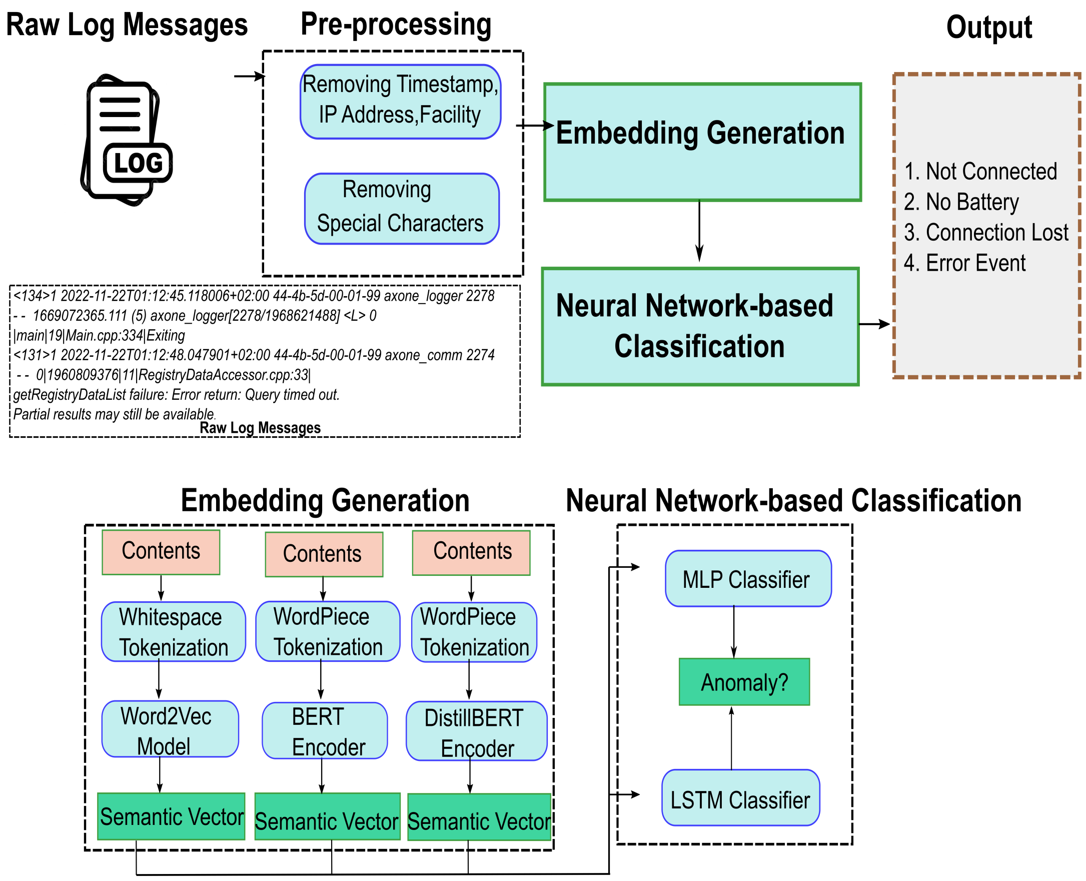
The extensive details on the theory of LLM compression are out of the scope of this research. However, knowledge distillation is used in this research because the original BERT model was computationally heavy for the resources and thus, its distilled version, DistilBERT was used. Therefore, this section presents some literature on knowledge distillation and empirically proven distilled models of BERT. There have been published works claiming that knowledge distillation is the best performance in terms of accuracy and computational among these four large models compression methods [37].

Knowledge Distillation [38] [39] is a compression technique in which a smaller model i.e. student model is trained to mimic the output of the larger model i.e. teacher model for the same given data set. Hinton et al. [39] in their paper discovered that the student model trained on a larger model generalized better than the same model trained from scratch. The reasoning they have put forward in the paper is that the teacher model gives extra information by showing some wrong predictions are more wrong than others. The student model learns through either offline distillation, online distillation, or self-distillation process. During the process of offline distillation the student model mimics a fine-tuned teacher model's performance whereas during online distillation, the training of student models runs together with the teacher model. During self-distillation, the student model learns from the previous version of itself.[40].

The student model mimics the teacher model based on three kinds of knowledge in general. They are response-based knowledge, feature-based knowledge, and relation-based knowledge. There have been many distilled models of BERT such as DistilBERT[41], TinyBERT [42], and MobileBERT [43]. DistilBERT has reduced the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster [41]. In the DistilBERT paper, they have introduced triple loss combining language modeling, distillation, and cosine-distance losses. The cosine-distance loss is used to align the hidden states of the student and teacher model. The DistilBERT model is a general-purpose pre-trained model that can be fine-tuned with good performances on downstream tasks, retaining the flexibility and modularity of larger models [41]. The downstream task of extracting feature embeddings from log lines in the large pool of log files is utilized in this research.

5 Implementation of NLP techniques using Machine Learning

This section describes the text vectorization techniques that were involved in this research. This chapter presents a detailed description of the dataset, labeling procedure, preprocessing techniques and tokenization methods that were implemented, and the methods of generating text embeddings. It also gives a detailed justification for the selection of certain algorithms for the process of tokenization and generating word embeddings. The system design of different NLP techniques generating word embeddings and the use of different machine learning algorithms to classify anomalous log events into designated labels is shown in Figure 22. As seen in Figure 22, raw log messages are preprocessed to remove high-cardinality fields like timestamps, and special characters before applying word embedding algorithms. Then the word embeddings are fed into a neural network-based classification model for the classification of log lines into different classes for the purpose of anomaly detection.



```
<134>1 2022-11-22T01:12:45.118006+02:00 44-4b-5d-00-01-99 axone_logger 2278
- -  1669072365.111 (5) axone_logger[2278/1968621488] <L> 0
|main|19|Main.cpp:334|Exiting
<131>1 2022-11-22T01:12:48.047901+02:00 44-4b-5d-00-01-99 axone_comm 2274
 - -  0|1960809376|11|RegistryDataAccessor.cpp:33|
getRegistryDataList failure: Error return: Query timed out.
Partial results may still be available.
```

Figure 22: Design framework for the implemented solution of anomaly detection in logs with NLP techniques

5.1 Raw Log Messages

This section gives details of the syslog dataset and its attributes used in this research. The format of the syslogs generated in GE Healthcare is already explained in Section 2.1.1. The syslog dataset is comprised of different log sources, such as hubs, virtual hubs, and patient monitoring viewer platforms. Each individual log file contains thousands of log lines with uneven formats. These log files are thus aggregated into a single file in a 'csv' format with the respective order of their presence in the source directory of storage. The log dataset consisted of normal and anomalous events where the number of anomalous events was rare compared to normal events thus making it a highly imbalanced dataset. These log lines are then subsequently labeled into different classes. The details of syslog data composition are given in bullet points below:

- Total number of log files - 150
- Total number of log lines - ~ 2.5 million
- Total number of classes - 5

The research work started with a large volume of log files and did not have any explicit strategy on how to approach the problem statement. The problem statement was to identify specific anomalous log events in a large pool of syslogs data as mentioned in Section 1. For this purpose, supervised machine learning was chosen for classification as the most appropriate method for classifying normal and anomalous log lines. For supervised learning, the first step is to label the data into different classes and thus labeling of the full dataset was carried out. A custom Python script was written for this process of labeling the log lines into different classes. We established a structure such that four different anomalous events were classified into four specific classes and a fifth class was assigned to incorporate all standard log events. There were 4 classes for specific keywords and one normal log class totaling 5 classes. The classes were labeled as 0, 1, 2, 3, 4.

- Log lines containing ASSOCIATED_CONNECTED: 4
- Log lines containing ASSOCIATED_CONNECTION_LOST: 3
- Log lines containing NOT_ASSOCIATED: 2
- Log lines containing ASSOCIATED_NO_BATTERY: 1
- Normal standard log lines: 0

These categories were assigned numerical identifiers ranging from 0 to 4 for simplicity and ease of processing. These log classes are expected to be expanded over the course of the study as there are plenty of other anomalous events with certain attributes. This structure of organizing logs into multi-tiered classifications according to their attributes provides a structured approach to navigating the area of solving the problem statement.

Sampling of Anomalous Syslog Lines

Sub-sampling is a strategic method employed when dealing with large datasets, especially in situations where computational resources are limited. It involves selecting a smaller, representative sample from the larger dataset to work with. The main benefit of sub-sampling is that it reduces computational complexity without losing significant information, making the processing more time and resource-efficient.

In our case, the implementation of sub-sampling facilitated the development of a proof-of-concept. This smaller dataset not only made the analysis more feasible given our resource constraints, but it also enabled us to refine the methodologies before applying them to the larger dataset.

From a statistical perspective, the key to successful sub-sampling lies in the distributional representation of the sample. It's crucial that the sub-sample maintains the variability and structure of the original data. Therefore, our Python script was carefully designed to extract log lines with labels as numbers 1, 2, 3, 4 ensuring that these chosen labels encapsulated the various log data anomalies of interest. By focusing on these labels, we were able to maintain the integrity of the data while reducing its size. Although the total number of log lines was reduced to 3240, this sub-sampled dataset provided comprehensive insights into the patterns and anomalies for the overall log data.

In the context of our log data, the traditional sub-sampling technique was not used but rather a stratified sampling was used to extract specific anomalous classes. This process of sampling does not fully capture the range of normal behaviors in log data, which could be a limitation in understanding the complete context of log events. So, it does not represent the true distribution of the log events real scenario. This potentially impacts the model's ability to distinguish between normal and abnormal events accurately. However, this sampling provided us with a manageable small dataset which was well enough to establish a proof-of-concept, ultimately paving the way for the optimization of our methodologies for the larger dataset.

5.2 Word Embedding Generation

After the dataset was labeled into distinct classes, it was necessary to convert these text corpus into numerical representation, often called with term "text vectorization". With the development of techniques such as text embeddings, the majority of NLP tasks have demonstrated superior results with the use of these embedding techniques. Prior to the concept of text embeddings, other techniques such as Bag-of-Words, Term Frequency-Inverse Document Frequency (TF-IDF), and Latent Semantic Analysis (LSA) were used. However, these methods had inherent limitations, including a failure to capture contextual information and semantics of words. Thus, embedding techniques such as Word2Vec and BERT and its distilled variant DistilBERT are implemented in this research work for generating embeddings.

Before converting the text into embeddings, it was necessary to preprocess each log line such that unnecessary elements can be removed from the log lines. The preprocessing step was kept simple, with an emphasis on maintaining the integrity of the data. The process of preprocessing was carried out using regex operators in Python.

The simple preprocessing involved the removal of the timestamp field along with the facility and version. The special characters such as square brackets ([]), curly braces (), parentheses (()), quotation marks (' '), and inverted commas(" "), semi-colons(;) were also removed during the course of preprocessing. These elements were deemed unnecessary as they did not contribute meaningful information for our purpose of anomaly detection. Finally, all the texts were converted into lowercase to eliminate the potential inconsistencies. The reason for not undergoing extensive preprocessing was based on several reasons. Primarily, excessive preprocessing can lead to the loss of crucial data thus distorting the true representation of original logs. Additionally, the complexity of the logs can make extensive preprocessing a time-consuming task, potentially slowing down the overall process without any significant improvement in the results. The extensive preprocessing was also tested in a separate dataset which severely slowed down the process and also did not offer any substantial benefit as the log files are not consistent with the format and use of whitespaces and characters. After preprocessing the log data, the processed log lines were forwarded for tokenization and subsequently for the generation of word embeddings. The different tokenization methods along with different text embedding methods implemented in experiments during this research work are presented in Section 5.2.1 and 5.2.2.

5.2.1 Word Tokenization and Word2Vec Experiments

As the dataset or number of log lines was relatively large for the available resources allocated for this research, the experimental framework was designed carefully. First, it was necessary to sub-sample the dataset into smaller units and proof-of-concept had to be delivered before processing the full dataset. For that purpose, a Python script was written to extract those log lines with labels from 1..4 from a large dataset to sub-sample and label it into 4 classes. These labels were the most interesting labels in the entire dataset. This process resulted in a fairly small dataset with 3240 log lines with 4 anomalous classes. The composition of a smaller sampled dataset is given below:

- *Total number of log lines in a sampled dataset: 3240*
- *Total number of log lines with class 1 : 241*
- *Total number of log lines with class 2 : 75*
- *Total number of log lines with class 3 : 787*
- *Total number of log lines with class 4 : 2141*

The dataset was comprised of data points and labels and stored in "CSV" format. The dataset was processed using the Pandas library [44] leveraging "DataFrame" feature in the library. Python regular expressions were used to remove the high-cardinality fields from the dataset. Then, the dataset was partitioned into training and testing sets in an 80/20 ratio, leveraging the scikit-learn [45] library, with a consistent random seed of 42 and stratification of the labels. This stratification ensured that even in the face of imbalanced data, each class was proportionally represented during the split. Setting a random seed to 42 ensures consistent data splits, thus facilitating the reproducibility of results. Prior to training the Word2Vec models, this partitioning method was crucial to prevent data leakage, thereby minimizing the risk of model overfitting and ensuring result consistency between training and testing sets.

Word2Vec Models for Word Embedding Generation:
Following the preprocessing of the dataset, the Word2Vec of family algorithms, CBOW, and skip-gram model were trained to generate word embeddings in an unsupervised manner. The algorithmic process of training the Word2Vec models is presented in Algorithm 1. Gensim [46], which is an open-source Python library for processing raw, unstructured digital texts ("plain text") using unsupervised machine learning algorithms was used to train the Word2Vec models. The output of a trained Word2Vec model is a set of word embeddings. Each word in the model's vocabulary is represented by a dense vector of fixed size. These vectors capture semantic information about words based on the context in which they appear in the training corpus. The embeddings were generated and those embeddings were used for downstream classification tasks to classify these 4 using machine learning classifiers.

Algorithm 1 Word2Vec Log Embedding

```
Require: Log lines $\{l : l \in L\}$
Ensure: Word embeddings $\{v_w : w \in W\}$
 1: Initialize WordEmbeddings as an empty list
 2: Initialize Word2VecModel from gensim.models
 3: for all log line $l$ in $L$ do
 4:     Preprocess $l$ to get $p_l$ (e.g., removing timestamps or other non-relevant data)
 5:     Tokenize $p_l$ to get words $w_l$
 6:     Add $w_l$ to the corpus for training
 7: end for
 8: Train Word2VecModel on the corpus
 9: for all log line $l$ in $L$ do
10:     for all word $w$ in $w_l$ do
11:         Pass $w$ to Word2VecModel to get $v_w$
12:         Append $v_w$ to WordEmbeddings
13:     end for
14: end for
15: return WordEmbeddings
```

5.2.2 Subword Tokenization and BERT/DistilBERT Encoder

Following the experiments with the static word embedding models (Word2Vec), the quality of the generated word embeddings was evaluated. Upon evaluation, it was found that the static word embeddings had certain limitations. Keeping this in mind, we experimented with contextual word embedding models which are more dynamic and preserve the semantic relationship between words used in different contexts. After the comparative study of various alternatives, the BERT pretrained model was chosen to generate word embeddings in the form of feature vectors. The selection was motivated by a trade-off between the model parameters and resources available to compute. As in Word2Vec experiments, similar preprocessing techniques were used to remove the high-cardinality fields and special characters from the data.

The tokenization process in the Word2Vec family of algorithms was carried out at word level to create word tokens using whitespace tokenization. Because of the inconsistent use of characters and spaces, it was essential to evaluate the performance of subword tokenization techniques to produce small subwords as tokens. The tokenizers splitting text int subword level are called trainable tokenizers. As BERT is built on Transformer encoder architecture as mentioned in Section 4.2, it was a perfect choice to encode the features into embeddings from preprocessed log lines.

Huggingface [47], which is a hub of open-source models for Natural Language Processing, computer vision, and other fields, was used to download pretrained BERT "bert_base_uncased" tokenizer and encoder to be applied to the input data. As in earlier experiments, the models were tested in a small dataset initially to generate embeddings from pretrained model and subword tokenizer. The process began with the input of each log line into a tokenizer (*"bert_base_uncased"*) after simple preprocessing steps. The same level of preprocessing for the log lines was done as in Word2Vec tokenization 5.2.1 with the removal of some special characters using regex operations in Python. As some of the log lines were very long sequences of words, a maximum length of 512 was used as a hyperparameter in the tokenizer to get tokens. Then obtained tokens were split of words into smaller units. The log lines are composed of error keys and codes, and other protocol codes, which are split into smaller units make the occurrences of new keys to be included in the vocabulary. Similarly splitting of words into sub-words made the embeddings robust to inconsistent use of spaces and characters in log lines. For example *"managerfile"* is treated as a familiar word if "mana", "##ger" and "##file" tokens are present in the vocabulary of subwords. These obtained subword tokens are passed into pretrained model 'bert_base_uncased' imported from Huggingface library. BERT pretrained model acted as a feature extractor to the ingested log data corpus. The outputs were obtained from the last hidden state of the transformer model as this is used as a feature extractor. The vectors obtained from the last hidden state is a tensor containing the final embedding vector of each token in the log lines. Then, the pooling operation is carried out using the Numpy library's mean function for all the vectors in the list. Pooling operation is a technique that takes the collection of vectors and condenses them into a single vector. By taking the mean, a single vector was created that represented the entire log line based on the embeddings of individual tokens. This operation was done to represent each log line as a single vector in a large log corpus, which can be considered as word embedding which represents the entire sentence. These embeddings are stored as numpy arrays which are used for downstream tasks.

As the BERT model has 110 million parameters thus making it computationally heavy even to use as a feature extractor. Thus it became infeasible to use it for a full dataset with 2.5 million log lines. Thus, model compression techniques were studied to reduce the model weights and parameters. It was found DistilBERT which is a distilled version of BERT produced comparable results to BERT. This pretrained model was already implemented in Huggingface [47] so the pretrained DistilBERT model from Huggingface along with a pretrained tokenizer was used for the full dataset to reduce the computational time. First, it was tested in a small dataset and after the comparable results, it was then applied to the full dataset. The preprocessing and tokenization followed the same steps as in the BERT pretrained tokenizer to obtain the encoded inputs. Then, similar to the BERT pretrained model, DistilBERT was fed with those encoded inputs and the process of obtaining the embeddings followed similar operations as in the BERT experiment. The embeddings generated from the DistilBERT in a full dataset was used for the downstream task of classification. Algorithm 2 shows the pseudo code for the implementation of using pretrained BERT/DistilBERT as a feature extractor to generate contextual word embeddings as semantic vectors

Algorithm 2 BERT/DistilBert Log Embedding

```
Require: Log lines $\{l : l \in L\}$, Pre-trained Bert/DistilBert model and associated
    Bert/DistilBert Tokenizer
Ensure: embeddings $\{v_l : l \in L\}$
 1: Initialize Embeddings as an empty list
 2: Initialize BertModel/DistilBert from pre-trained BERT/DistilBert
 3: Initialize BertTokenizer/DistilBert associated with the pre-trained BERT/DistilBert

 4: for all log line $l$ in $L$ do
 5:     Preprocess $l$ to get $p_l$ (e.g., removing timestamps or other non-relevant data)
 6:     Tokenize $p_l$ using BertTokenizer/DistilBertTokenizer to get tokens $t_l$
 7:     Convert $t_l$ to InputIDs $i_l$ with padding and truncation
 8:     Pass $i_l$ to BertModel/DistilBert to get HiddenStates $h_l$
 9:     Calculate $v_l$ as the mean of $h_l$
10:     Append $v_l$ to Embeddings
11: end for
12: Form a matrix $X$ whose columns are semantic Embeddings, and let $u$ be its first
    singular vector
13: for all $v_l$ in Embeddings do
14:     $v_l \leftarrow v_l - uu^T v_l$
15: end for
16: return Embeddings
```

5.3 Neural Network-based Classification

After the process of generation of word embeddings from the data corpus, machine learning downstream tasks were performed for the purpose of anomaly detection using classification models. Open-source libraries like Numpy, Pandas, and TensorFlow with Keras [48, 49] were used. The experiments were conducted on two computers with processors operating at individual levels. The CPU capabilities were 32 GB and 64 GB of RAM. The Anaconda [50] was used as a virtual environment which helped to manage the experiments. Similar to word embedding experiments, machine learning experiments were conducted in two separate datasets and their respective embeddings. As the experiments were conducted with relatively constrained computation resources, the experiments had to be crafted carefully. The machine learning experiment was started with a simple Multi Layer Perceptron (MLP) as a base classifier. The MLP classifier achieved good performance with Word2Vec embeddings. The MLP classifier was then followed by the use of Long Short Term Memory (LSTM) models. The literature on log analysis and anomaly detection performed in public datasets showed the extensive use of LSTM networks for better classification results.

5.3.1 Multi Layer Perceptron Classifier

Multi Layer Perceptron (MLP) was used as a base classifier to check the quality of word embeddings generated from Word2Vec models. TensorFlow [48] version 2.10 was used throughout this research work. MLP as a classifier was used only in a sampled dataset with 3240 log lines. This is motivated by the hypothesis that MLP would act as base classifier which would be a stepping stone towards more sophisticated neural network architectures. MLP is a simple neural network architecture only composed of Dense layers connected in a feed-forward network. The input layer was a Dense layer with 128 nodes or neurons. As the embeddings generated from Word2Vec were of dimension 100, the input shape of embeddings was assigned as 100. Activation function 'relu' was used. The hidden layer was again the Dense layer with 64 nodes with 'relu' activation function. Finally, the Dense layer with 4 output nodes was assigned with the softmax function.

The MLP model was trained using a cross-entropy loss function with adam optimizer. The model was trained for 20 epochs with a batch size of 16 in a training corpus with the corresponding labels in a supervised manner. The trained model was thus evaluated in a test corpus. The test corpus was converted into embeddings using the trained Word2Vec model from the training corpus. The trained Word2Vec model acted as a feature extractor for the test data to be evaluated during inference. The pseudo-code used for the MLP classifier is presented in Algorithm 3.

Algorithm 3 MLP Classification

```
Require: Training vectors $X_{\text{train}}$, Training labels $Y_{\text{train}}$, Testing vectors $X_{\text{test}}$
Ensure: Predicted labels $Y_{\text{pred}}$
1: Define MLP architecture: Dense layer with 128 units (ReLU activation), Dense
   layer with 64 units (ReLU activation), and a Dense layer with 4 units (softmax
   activation)
2: Fit MLP model on $X_{\text{train}}$ and $Y_{\text{train}}$ for a specified number of epochs
3: Predict $Y_{\text{pred}}$ on $X_{\text{test}}$
4: return $Y_{\text{pred}}$
```

5.3.2 Long Short Term Memory Classifier

The generated embeddings were then used for downstream classification tasks with the LSTM network. A shallow LSTM network was used because of an imbalanced and relatively small number of samples for anomaly classes. By opting for a shallow LSTM, the model's complexity was effectively limited, reducing the risk of overfitting in the context of the small dataset. Shallow LSTMs have fewer layers and parameters, making them less prone to memorizing noise in the training data, and promoting better generalization to the validation and test set.

The number of layers and cells used in all the experiments was consistent throughout all the experiments carried out during the research. This was mainly motivated for the purpose of comparison of the results from static and contextual embeddings mainly Word2Vec and BERT pretrained models and its distilled version DistilBERT. TensorFlow [48] version 2.10 was used for all the experiments.

The number of LSTM cells in the input layer was 128 and the number of cells in the hidden layer was 64 with the output Dense layer consisting of 4 or 5 layers depending on the number of classes to be classified with a softmax function. On a small sub-sampled dataset, the output layer was composed of a Dense layer with 4 neurons whereas the full dataset had 5 neurons with softmax output function. The layers were stacked in a sequential manner.

The input to the LSTM using word embeddings generated from Word2Vec was of dimension (1, 100) as the embeddings generated consisted of 100 dimensions. The LSTM model was trained with adam optimizer with categorical_crossentropy as a training loss function. Adam optimizer was chosen for this task because of its low memory requirements and less effort in initializing and tuning of learning rate. Word2Vec model for word embeddings small dataset was mainly purposed for creating a base classifier using MLP and LSTM networks. Adam performed exceptionally well for the task so the choice of adam was also continued for the training with a full dataset. However, other optimizers such as Stochastic Gradient Descent (SGD) with tuning a learning rate could be experimented with to compare and evaluate performance as a future work. The model was trained for 30 epochs for a small sampled dataset for the embeddings generated from CBOW and skip-gram Word2Vec models. The training process underwent 20 epochs for the embeddings generated from the DistilBERT model.

After the model training was completed, the test set was evaluated with the obtained trained LSTM model. First, the raw test data went through the same preprocessing steps like removing some unwanted characters and timestamps as in training data. Then the test data was fed to the trained Word2Vec model to obtain test embeddings which can also be termed test set vectors. The trained Word2Vec model acted as a feature extractor for the test data which was fed to the trained LSTM model. The pseudo-code for the LSTM classifier is presented in Algorithm 4.

Algorithm 4 LSTM Classification

```
Require: Training vectors $X_{\text{train}}$, Training labels $Y_{\text{train}}$, Testing vectors $X_{\text{test}}$
Ensure: Predicted labels $Y_{\text{pred}}$
1: Standardize $X_{\text{train}}$ and $X_{\text{test}}$
2: Reshape $X_{\text{train}}$ and $X_{\text{test}}$ to three dimensions for LSTM input
3: Define LSTM architecture: LSTM layer with 128 units (return sequences), LSTM
   layer with 64 units, and a Dense layer with 4 units (softmax activation)
4: Fit LSTM model on $X_{\text{train}}$ and $Y_{\text{train}}$ for specified epochs
5: Predict $Y_{\text{pred}}$ on $X_{\text{test}}$
6: return $Y_{\text{pred}}$
```

6 Experiments

This chapter presents the results of all the experiments conducted during this thesis research. The experiments involved multiple algorithms for generating word embeddings and multiple machine learning models for the classification of those word embeddings in a supervised method. Section 6.1 gives insights on evaluation metrics and section 6.2 presents the results in numerical format along with the confusion matrix with all the metrics described in Section 6.1.

6.1 Evaluation Metrics

In the domain of machine learning and data analytics, accuracy is frequently used as a primary metric for evaluating the performance of classification models. However, its utility becomes limited in the case of imbalanced datasets—where one class significantly outnumbers the other classes. Specifically, for datasets where the majority class dominates, a naive classifier predicting solely the majority class can yield high accuracy, even if it entirely misclassifies the minority class. Such a scenario is not merely hypothetical as it is the case with log anomalies where the occurrence of a single anomalous event is seen in hundreds or thousands of normal log events. Relying solely on accuracy in these cases obscures the model's failures, rendering it an unreliable and potentially misleading metric. Thus, for imbalanced datasets, it is imperative to incorporate more granular metrics, such as precision, recall, and the F1 score, which offer a nuanced understanding of a model's performance across multiple classes. To evaluate these metrics we have used a confusion matrix for the performance of a machine classifier in a test dataset. Some of the terminologies used for the calculation of these evaluation metrics are as follows:

1. True Positive (TP): When a prediction is both correct and positive, it is referred to as a true positive.
2. False Positive (FP): When a prediction is incorrect but still positive, it is referred to as a false positive.
3. True Negative (TN): When a prediction is both correct and negative, it is referred to as a true negative.
4. False Negative (FN): When a prediction is incorrect and negative, it is referred to as a false negative.

Based on these technical specs, the evaluation metrics such as accuracy, precision, recall, and F1-score are calculated. The brief introduction of aforementioned the evaluation metrics are as follows:

► **Accuracy:** Accuracy is the ratio of the total number of correct predictions to the total number of images in a dataset. The total number of correct predictions is calculated by summing the true negatives (TN) and true positives (TP). In mathematical notation, accuracy (AC) can be expressed as:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{24}$$

The ideal value for accuracy is 1, indicating perfect predictions, while the lowest value is 0, indicating completely incorrect predictions.

► **Recall:** Recall is the ratio of true positive predictions to the total number of positive predictions. The total number of positive predictions is calculated by summing the true positives (TP) and false negatives (FN). Therefore, the formula for recall can be expressed as:

$$Recall = \frac{TP}{TP + FN} \tag{25}$$

The ideal value for recall is 1, indicating that all positive instances are correctly identified, while the lowest value is 0, indicating that no positive instances are correctly identified.

► **Precision:** Precision is the ratio of true positives to the sum of true positives and false positives. Therefore, the formula for precision can be expressed as:

$$Precision = \frac{TP}{TP + FP} \tag{26}$$

The ideal value for precision is 1, indicating that all positive predictions made by the model are correct, with no false positives, while the lowest value for precision is 0, indicating that there are no correct positive predictions, resulting in only false positives.

► **F-measure:** F-measure (F1-score) combines precision and recall into a single value to indicate the performance of the model.

$$F1 - score = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{27}$$

The ideal value for F-measure (F1-score) is 1, indicating that the model achieves an optimal balance between precision and recall, resulting in the best possible overall performance.

6.2 Results

Recalling from Section 5, the full dataset used in this research consisted of ~ 2.5 million log lines, with around ~ 3240 anomalies (i.e., four classes of anomalies), while the remaining log lines were considered normal. This resulted in a total anomaly distribution of 0.1261%. This kind of distribution is commonly encountered in regular log streaming sessions in real-world scenarios. Therefore, as explained in Section 5.2.1, our first step involved analyzing a subset of samples containing ~ 3240 anomalies. The objective was to classify the four different classes of anomalies in a sampled dataset and select the best-performing model with a trade-off between performance and computational cost. Once the selection of the suitable model was completed, we would then proceed to train the model on the full dataset consisting of 2.5 million log lines with the best-performing model. The goal was to classify the full dataset into five labels, where class 0 is attributed to the normal category.

Comparative Study of Best-performing Word Embedding Model

In order to evaluate the effectiveness of various word embedding techniques, we conducted a comparative study on a smaller, sampled dataset. Different combinations of word embedding methods and a constant discriminator/classifier were utilized in the experiments. The rationale behind such a methodology was to understand the distinct ways in which these methods generate word representations, and how these disparities impact the performance of the classifier. Different word embeddings capture semantic and syntactic relations between words in unique ways, which can dramatically influence the outcome of the task. Therefore, comparing embeddings created by different algorithmic methods offers insights into techniques that are most suited for the specific application.

In our study, we employed a variety of word embedding techniques, including Word2Vec with CBOW and Skip-Gram variants, BERT, and DistilBERT. In all the cases, we kept the discriminator constant, specifically MLP, while changing the word embedding models. This strategy allowed us to eliminate potential confounding factors, thereby enabling a more direct and isolated assessment of the impact each word embedding method had on the overall model performance. The preprocessing steps remained consistent across all the word embedding models. We implemented a simple preprocessing procedure to ensure that any variations observed in the model performance could be attributed solely to the differences in the word embedding methods and not to any discrepancies in preprocessing.

This comparative study was conducted in a smaller sampled dataset before being implemented on a full dataset with a large volume of log data. The process of training machine learning models, especially with vast volumes of data, can be both time-consuming and demanding in terms of computational resources. By initially assessing the performance of different word embedding techniques on a smaller scale, we were able to identify the most effective methods. This approach, in turn, saved significant time and computational resources when scaling up to larger datasets. It ensured an efficient and cost-effective strategy for the overall machine-learning process, which was critical given the complexities associated with handling large datasets.

We evaluated the performance of four different combinations of embedding generation and machine learning classifiers on the subset of ~ 3240 anomalies. We present our evaluation result for these combinations:

► **Combination 1: Word2Vec (CBOW) as Embedding Generator + MLP Classifier**:
After the sampling of 3240 log lines of 4 classes and preprocessing of those log lines, word embeddings were generated using the Word2Vec algorithm (CBOW) model as presented in Algorithm 1. First, we performed a train/test split of 80/20 ratio to split the separate dataset into separate train and train sets. Then, simple prepossessing was applied and the training dataset was trained using the CBOW variant of Word2Vec model. The experimental details of the Word2Vec model is already presented in 5.2.1. The generated word embeddings were trained using the MLP classifier with a softmax loss function. The results obtained on a test set using this combination is presented as a confusion matrix in Table 1.

It is well known that a confusion matrix is used in machine learning and statistics to understand the performance of a classification model. It shows the true positive, false positive, true negative, and false negative values for the model predictions compared to the actual class labels. Here in Table 1, the rows represent the actual classes, while the columns represent the predicted classes of a classification model. For a class X, the entry at row X, column X gives the count of true positives (TP), i.e., the number of instances correctly identified as class X. Entries outside the main diagonal give the count of misclassifications. The diagonal entries represent the correct classifications for each class and this structure remains consistent throughout all the experiments. It can be observed in Table 1 that 42 instances were correctly identified (True Positives) while 6 instances were incorrectly classified for class label 1. 13 instances were correctly identified for class label 2 while 2 instances were misclassified as class 3. Similarly, 134 instances out of 158 instances were correctly identified for class 3 while 24 of them were misclassified. Here, 4 instances were misclassified as class 1 and 4 instances as class 4. Similarly, for class 4, 364 instances were classified correctly out of 428 instances. Here, 63 instances were classified incorrectly as class 2, and 1 instance was misclassified as class 3. We also observe a combined macro-averaged precision of 0.75, a macro-averaged recall of 0.83, an overall accuracy of 0.83, and a macro-averaged F1 score of 0.72. In summary, the model performed well for class 1, class 3, and class 4 but struggled to correctly identify class 2. This is indicative of the minority class being harder for the model to distinguish, possibly due to them being less represented in the training data.

Table 1: Confusion matrix for Combination 1

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**43**	5	0	0
Class 2	3	**12**	0	0
Class 3	0	24	**120**	14
Class 4	0	63	1	**364**

► **Combination 2: Word2Vec (Skip-gram) as Embedding Generator + MLP Classifier**:

Next, the experimental study was done with the combination skip-gram variant of Word2Vec with MLP classifier. All the parameters and hyperparameters were kept the same as in the earlier experiment of Word2Vec with the CBOW model with an addition of *"sg=1"*. The ratio of train/test split and preprocessing steps remained the same as in combination 1. The word embeddings that were generated from the Word2Vec skip-gram model were trained using an MLP classifier with a softmax function in the output layer. The parameters and hyperparameters in MLP also remained identical as in combination 1. Our result using this combination is presented as a confusion matrix in Table 2.

As we can see in Table 2, 41 instances were correctly identified (True Positives) for class 1 while 7 instances were incorrectly classified. Among the misclassifications, 5 instances were misclassified for class 2, and 2 instances were misclassified for class 3. For class 2, 11 instances were classified correctly while 4 instances were misclassified. Similarly, 133 instances out of 158 instances were correctly identified for class 3 while 25 instances were misclassified. Here, 24 instances were misclassified as class 2, and a single instance was misclassified as class 1. Similarly, for class 4, 364 instances were classified correctly out of 428 instances. Here, 63 instances were misclassified as class 2, and 1 instance was misclassified as class 3. We observed a macro averaged precision of 0.74, recall of 0.85, F1-score of 0.74, and an overall accuracy of 0.85.

Table 2: Confusion matrix for Combination 2

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**41**	5	2	0
Class 2	4	**11**	0	0
Class 3	1	24	**133**	0
Class 4	0	63	1	**364**

► **Combination 3: BERT (Subword Tokenization) as Embedding Generator + MLP Classifier**:

After the experiments with static word embedding models utilizing Word2Vec for both CBOW and skip-gram models, we proceeded to further experiments with contextual embedding models. For this, we used the Bidirectional Encoder Representations from Transformers (BERT) pretrained model to generate word embeddings. The experiments conducted with the Word2Vec family of algorithms resulted in numerous misclassifications, prompting us to explore a more advanced, context-aware BERT model. A detailed explanation of the BERT experimental setup has been already presented in Section 5.2.2, while the outcomes are presented in the form of a confusion matrix in Table 3.

As seen in Table 3, the combination of a BERT model as an embedding generator and MLP as a classifier achieved almost perfect results in all of the evaluation metrics. This implies that BERT generates rich context-aware feature embeddings. However, this came with a price of computational cost. As BERT pretrained model has 120 million parameters, the process of feature extraction was a slow process. With the available computation resources in our hands, it took almost 46 seconds to process a batch of 128 log lines. This turned out to be a significant hurdle which acted as a bottleneck for deployment as the computational resources should be feasible along with the model's performance.

Table 3: Confusion matrix for Combination 3

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**48**	0	0	0
Class 2	0	**15**	0	0
Class 3	0	0	**158**	0
Class 4	0	0	0	**428**

► **Combination 4: DistilBERT as Embedding Generator + MLP Classifier**:
The DistilBERT pretrained model was identified as an alternative to the computationally demanding BERT model without compromising performance. Then, MLP was used as a discriminator/classifier model was used to classify the classes. The embeddings from DistilBERT were generated using similar preprocessing and mean pooling steps as in BERT in the earlier experiments. The results are presented in the confusion matrix in Table 4.

We observed impressive results in an experimental study with a combination of DistilBERT to extract embeddings/features and LSTM classifier. All the instances of class 1, class 3, and class 4 were classified correctly. There were 5 misclassifications out of 15 instances for class 2 where they were misclassified as class 3. We observed a macro average precision of 0.99, recall of 0.92, accuracy of 0.99, and an F1 score of 0.95. The misclassification occurred with the log class which had the fewest training examples which is justified in supervised learning. The computational cost also aligned with the paper [41] as it took an average of 22 seconds to extract embeddings for 128 log lines. This was consistent with the authors' claim that DistilBERT is 60% faster retaining 97% of the BERT's performance.

Table 4: Confusion matrix for Combination 4

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**48**	0	0	0
Class 2	0	**10**	5	0
Class 3	0	0	**158**	0
Class 4	0	0	0	**428**

Selection of Best Performing Word Embedding Model

The comparative evaluation of word embedding models revealed distinctive strengths and drawbacks of Word2Vec, BERT, and its lightweight version, DistilBERT. Word2Vec with both of its variants was computationally effective and also produced satisfactory results with an accuracy of approximately 85 percent per class.

However, for the problem statement of automated log analysis, this level of performance was not sufficient for production and deployment. Further, Word2Vec models encountered problems with Out-of-Vocabulary (OOV) words. As we move towards larger datasets, the number of OOV words can be expected to increase, potentially leading to a regression in performance. While BERT emerged as the most robust model in terms of performance, it presented significant computational demand, a factor that could not be overlooked in the context of processing larger datasets. On the other hand, DistilBERT proved itself to be an effective alternative to BERT. Despite its smaller size and faster computational speed, DistilBERT managed to retain the strong performance characteristics of its teacher model, BERT. Considering these observations, it became evident that the DistilBERT struck the right balance between performance and computational efficiency. Therefore, it was chosen as a preferred word embedding generator for future experiments involving the full dataset.

Experiments for Selection of the Best Performing Classifier

Following the selection of DistilBERT as our primary word embedding model, the next stage of our research involved evaluating the performance of various machine learning classifiers. The goal of this study was to determine the most effective classifier for distinguishing between normal operations and different types of anomalies in log data. For this, we utilized DistilBERT-generated embeddings as a fixed input across all tests, focusing our variations on the classifier. This contrasts with our previous approach, where the word embedding model was changed whereas the classifier model remained constant.

The classifiers chosen for evaluation were a mix of simpler models, including Logistic Regression, and MLP, and more sophisticated models such as the LSTM network. The use of simpler and more sophisticated models was to understand how changes in the complexity of the classifier affected the performance of our system. This approach could offer insights into the model complexity of the classifier and its performance in terms of accuracy, precision, and recall. The results of this comparative study, which were obtained by varying the classifier while keeping the word embeddings model constant, are presented in the following section under the names of the respective combinations used.

► **Combination 5: DistilBERT as Embedding Generator + Logistic Regression Classifier**:

Logistic Regression is the simplest machine learning classifier used in the field of machine learning. The theoretical details of logistic regression can be found in [51]. The use of Logistic Regression in this research is to understand how the classifier model impacts the classification of generated word embeddings. As in MLP, Logistic Regression as a classifier was used only in a sampled dataset with 3240 log lines. The train and test split ratio of the dataset was the same as in MLP.

The combination of DistilBERT with Logistic Regression demonstrated varying performance across the different classes. For Class 1, the model successfully predicted 23 instances correctly. However, it also misclassified 3 instances as Class 2 and 22 instances as Class 4. In the case of Class 2, the classifier failed to classify any instances, instead, incorrectly classifying all 15 instances as Class 4. The model's performance was better with Class 3, correctly classifying 136 instances out of 158 instances, but also incorrectly predicting 22 instances as Class 2. For Class 4, the model showed better performance, correctly predicting 428 instances, while incorrectly classifying 54 instances as Class 2. In summary, while the DistilBERT and Logistic Regression combination showed good performance for Class 3 and Class 4, it struggled with Class 1 and Class 2. The results as a confusion matrix are shown in Table 5. We obtained a macro-averaged precision of 0.68, recall of 0.73, accuracy of 0.82, and F1-score of 0.7. Accuracy is not always the right metric for multi-class classification in an imbalanced dataset as we can see in the results. Per-class accuracy of class 2 is 0, however the overall accuracy is 82 %.

Table 5: Confusion matrix for Combination 5

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**23**	0	3	22
Class 2	0	**0**	0	15
Class 3	0	22	**136**	0
Class 4	0	54	0	**374**

► **Combination 6: DistilBERT as Embedding Generator + LSTM Classifier**: The combination of DistilBERT and Logistic Regression did not produce satisfactory results in our experiment. The experiment showed that the performance of Logistic Regression was inferior when compared to the combination of DistilBERT embeddings and MLP classifier which was previously tested during the word embeddings assessment phase. This outcome was somewhat expected, given that Logistic Regression is a simpler model compared to MLP. This led us to explore a more advanced classifier, namely the Long Short-Term Memory (LSTM) model. The results of the experiment involving the DistilBERT + LSTM combination are presented in Table 6.

We can see impressive results with a combination of DistilBERT features and the LSTM classifier. All the instances of class 1, class 3, and class 4 were classified correctly. There were 2 misclassifications out of 15 instances for class 2 where they were misclassified as class 4. We observed a precision of 0.99, a recall of 0.9787, an accuracy of 0.99, and an F1 score of 0.98. The computational cost also aligned with the paper [41] as it took an average of 22 seconds to extract embeddings for 128 log lines. This was consistent with the authors' claim that DistilBERT is 60 % faster retaining 97 % of the BERT's performance.

Table 6: Confusion matrix for Combination 6

Actual/Predicted	Class 1	Class 2	Class 3	Class 4
Class 1	**48**	0	0	0
Class 2	0	**13**	0	2
Class 3	0	0	**158**	0
Class 4	0	0	0	**428**

Selection of Best Performing Classifier and Implementation in Full Dataset

The comparative study of classifier models was conducted in embedding generated from DistilBERT using Logistic Regression, MLP, and LSTM and results are presented in confusion matrices in Table 4, 5, 6. Based on the results, Logistic Regression, the simplest of the three models, performed notably worse, possibly due to its inability to capture complex feature relationships in the data. On the other hand, both the MLP and LSTM models showed significant improvements in performance, performing nearly on par with each other. This indicates that they are both capable of capturing more intricate patterns in the data.

However, the LSTM model was chosen as the superior model despite this. The ability of LSTM ability to retain sequential information is critical, especially considering the nature of text data where order matters. This sequential understanding provides additional context which can enhance model performance and robustness. Moreover, as we consider scaling up to larger datasets and adding additional features such as timestamps as a part of future work, the LSTM's capacity to handle sequence data could become even more valuable. MLP, while it performed well on this dataset, may not handle these challenges as effectively due to its lack of sequence understanding. It assumes that inputs are independent of each other, which may not be the case with larger datasets and timestamped data. In conclusion, while both MLP and LSTM showed strong performance, the LSTM's potential for scalability and enhanced handling of sequential data made it the preferable choice for this task and for possible future extensions of the work.

After the selection of a suitable word embedding model and classifier, we progressed to apply the suitable combination to our full dataset, which presents a more complex task due to its increased volume and number of classes. The dataset is heavily skewed, with normal logs accounting for approximately 99.87 % of the total log lines. This imbalance in classes represents a challenging condition for machine learning models.

Initially, we applied our selected model to five classes, including the four previously tested classes and a new class representing normal logs. This setup allowed us to evaluate the performance of our chosen combination of DistilBERT and LSTM under conditions and data distributions closer to real-world scenarios. Subsequently, we expanded our system further by introducing an additional class, allowing us to examine the scalability of our selected approach and its effectiveness when handling an even greater number of classes.

► **Combination 7: DistilBERT as Embedding Generator + LSTM Classifier on a full dataset**:

From our series of experiments, it was proven empirically that DistilBERT was the most suitable balance between performance and computational efficiency. Therefore, we chose DistilBERT as our primary feature extractor for the task at hand. Combined with the LSTM classifier, which has a proven ability to effectively process sequential data and capture long-term dependencies, the pairing turned out to be a powerful and resource-efficient solution. We trained our final model on a full dataset, consisting of the DistilBERT encoder and LSTM classifier. The results are shown in Table 7.

In this experiment, class 0 was assigned to the normal log events which represented 99.8739 % of the log lines. This was an enormous class imbalance in the dataset. Despite a huge imbalance in the dataset, performance aligned with the small sampled dataset. As we can see in Table 7, all the instances of class 0 which represented the normal log events were predicted correctly. There was 1 misclassification out of 46 instances of log lines belonging to class 1 predicting 45 correctly. There was relatively dismal performance in class 3 where the trained model predicted 10 correct instances out of 15 and 5 instances were misclassified as class 4. For class 3, 157 instances were classified correctly and 1 instance was predicted wrongly to class 2. All the instances of test log lines for class 4 were predicted correctly. We obtained macro-averaged precision of 0.99, recall of 0.92, F1-score 0.95, and accuracy of 0.999.

Table 7: Confusion matrix for Combination 7

Actual/Predicted	Class 0	Class 1	Class 2	Class 3	Class 4
Class 0	**512846**	0	0	0	0
Class 1	0	**45**	1	0	0
Class 2	0	0	**10**	0	5
Class 3	0	1	0	**157**	0
Class 4	0	0	0	0	**428**

Scaling the number of classes
As already discussed, generating word embeddings is an unsupervised task and is independent of the class labels. This approach ensures that the focus remains on representing the raw data as effectively as possible, which provides a degree of scalability to the system. In our experiment, we initially created embeddings for our dataset, disregarding the class labels, and saved the obtained embeddings in a *'.npy'* file. We wanted to test if the model could effectively handle an increase in the number of classes so that the same embeddings could be used for a variety of downstream tasks. Hence, we introduced an additional class, which was initially a part of the 'normal' log class and labeled as '0'. This new class was designated as '5' and it was identified as *'error event'*.

► **Combination 8: DistilBERT as Embedding Generator + LSTM Classifier on a full dataset with 6 classes**:
Table 8 shows the confusion matrix of 6 classes where the word embeddings generated from DistilBERT are fed into an LSTM classifier. We can see that all the instances of class 0 which represented the normal log events were predicted correctly. There were 2 misclassifications out of 48 instances of log lines belonging to class 1 predicting 46 instances correctly and 2 misclassified as class 1. As in combination 7, there was relatively dismal performance in class 2 where the trained model predicted 9 correct instances out of 15 where 3 instances were misclassified as class 2, and 3 instances were misclassified as class 4. For class 3, all 157 instances were classified correctly. For class 4, 427 instances were predicted correctly out of a total of 428 instances. All the instances of test log lines for class 5 were predicted correctly. We obtained a macro-averaged precision of 0.96, recall of 0.93, accuracy of 0.99, and F1-score of 0.94.

Table 8: Confusion matrix for Combination 8

Actual/Predicted	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5
Class 0	**512308**	0	0	0	0	0
Class 1	0	**46**	2	0	0	0
Class 2	0	3	**9**	0	3	0
Class 3	0	0	0	**157**	0	0
Class 4	0	0	1	0	**427**	0
Class 5	0	0	0	0	0	**539**

Summary of Results:

A comprehensive experiment of eight distinct combinations of word embedding methods and machine learning classifiers was conducted and results were obtained which are presented above in the confusion matrix with the evaluation metrics. Initially, we began by identifying the best and most suitable word embedding model through a comparative evaluation study conducted on a smaller dataset, while keeping the classifier constant. Subsequently, we shifted our focus to identifying the optimal classifier, maintaining the selected word embedding model as constant during the experiments.

The combined assessment, as depicted in Figure 23 and Figure 24, exhibits the performance of each combination based on accuracy, F1-score, precision, and recall metrics. Although the BERT model as the embedding generator produced ideal results, it was important to consider the computational demands of this setup. BERT, while delivering excellent performance, was computationally heavy, potentially impacting its deployment in resource-limited environments. So, DistilBERT was selected as a more efficient and effective alternative to BERT. DistilBERT, being a lighter version, nearly retained the performance of BERT while significantly reducing computational requirements. The combination of DistilBERT as an embedding generator and LSTM as a classifier was the ideal choice in terms of both performance and computational efficiency.

It can be seen that contextual embeddings like BERT/DistilBERT outperformed the static embedding technique, Word2Vec, in our experiments. This can be attributed to the Transformer-based architecture and attention mechanisms inherent to BERT/DistilBERT. These models encode context into their representations by applying different levels of attention to all the words in the context, thus providing a richer representation of the input text. Additionally, BERT/DistilBERT utilizes subword tokenization, which has turned out to be effective in handling out-of-vocabulary (OOV) words, a common issue in natural language processing. By breaking down words into smaller subword units, these models can construct representations for new words based on their constituent subwords. This helps in better generalization to unseen/unknown data and also improves the robustness of the model against the limitations of vocabulary, especially in a log data context where new terminologies can frequently occur.

For the classifier, The LSTM classifier demonstrated superior performance over Logistic Regression and MLP. This is due to LSTM's ability to capture long-term dependencies in sequential data, which is crucial for analyzing the context of log data. LSTM's inherent capacity for handling sequences makes it suitable for our task and future work where timestamps need to be taken into account. With this optimal combination of DistilBERT and LSTM, we performed experiments in a full dataset with 5 classes and later scaled to 6 classes.

Once we scaled to the full dataset with a combination of DistilBERT and LSTM, we found results remained consistent with impressive performance. This was achieved amidst a significantly skewed class distribution where approximately 99.87% of the log lines belong to the 'normal' category. Despite such an imbalance, our chosen combination demonstrated the capacity to effectively handle this issue, maintaining good performance across the classes. However, we also observed that for certain classes where the training sample size was relatively small, the performance appeared weaker, as illustrated in the confusion matrices (Tables 6, 7, and 8). Despite this, the overall performance of the chosen combination confirmed its suitability for the task at hand.

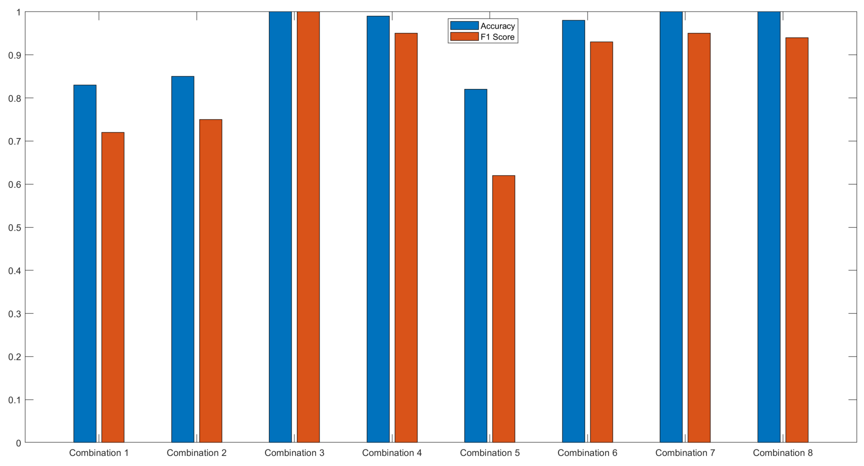


Figure 23: Accuracy and F1-score for all 8 combinations

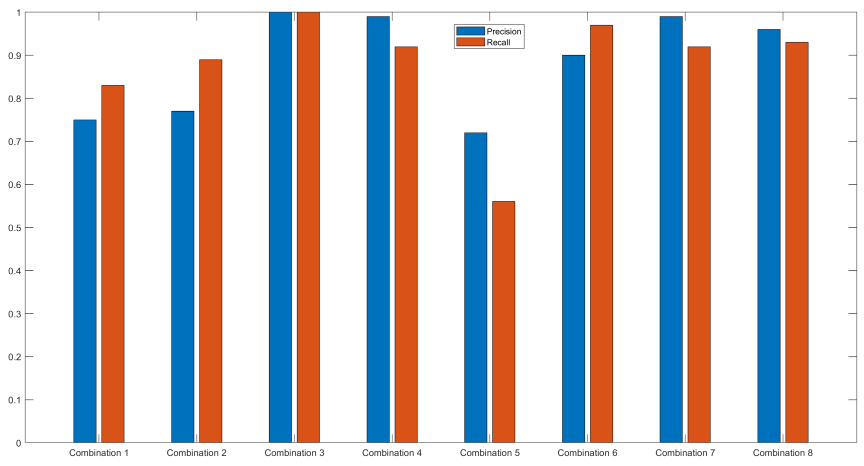


Figure 24: Precision and Recall for all 8 combinations

7 Conclusion

The main objective of this thesis work was to detect anomalies in the large volume of syslog data using machine learning and natural language processing techniques. To perform log analysis, an Elastic stack comprised of Elasticsearch and Kibana was studied and implemented to perform exploratory data analytics of syslogs. Kibana was used to visualise the results of analytics and queries performed during the study of statistical distribution of various keywords and sources of log events. Elastic agent was used as a shipping agent and parsing tool to ingest data into Elasticsearch. Elastic Stack served as an effective tool to get numerical insights from the data prior to applying machine learning for the purpose of log analysis and anomaly detection. Elastic stack also served as a multi-purpose tool that could be used by the Continuous Integration/Continuous Development (CI/CD) team as a log management and analysis tool.

Log analysis and anomaly detection were carried out by using machine learning and natural language processing techniques. First, the dataset was labeled into different anomalous classes and normal classes. Then, the dataset was sampled into a small version with only 4 classes of anomalies. Then, static embeddings and contextual embedding algorithms were applied to generate word embeddings. For static embedding, two Word2Vec algorithms, namely CBOW and skip-gram models were used. The generated embeddings were used for the downstream task of classification using Logistic Regression, MLP, and LSTM networks. For contextual embeddings, BERT and DistilBERT were implemented to select the best-performing model with a trade-off between performance and computational complexity. DistilBERT with LSTM classifier was chosen as a suitable combination to perform the task of log anomaly detection in the full log dataset. The classification results of detection of anomalous events with our approach achieved overall accuracy in excess of 0.99 with the macro averaged precision of 0.96, recall of 0.93, and F1 score of 0.94 in a comparatively large dataset with approximately 2.5 million log lines and 6 classes.

This research offers multiple benefits for the organization for which this thesis work was conducted for log management and the detection of anomalies in logs. The Elastic Stack integrated during this study could function as a centralized logging unit for a company's Continuous Integration/Continuous Delivery (CI/CD) team. It provides an enhanced visualization and analysis tool compared to existing solutions in effectively analyzing log data. The Elastic Stack, as used in this research for exploratory data analysis and visualisation tool can be used in other data analysis and machine learning experiments. Our experimental results are promising with high performance in all of the evaluation metrics. The high performance can be attributed to a good embedding model where it could capture the high-level representation of words in log files. There was limited variability in our dataset which made the model easier to learn which was also another cause for good performance metrics in the test data. This is however one of the limitations of this thesis work. But, on the other hand, it shows that the problem can be tackled and more complicated problems can be tackled in the future using a similar approach. Despite this, it is important to note that the current model will need further improvements before it can be fully deployed in a production environment for complete automation of log analysis and anomaly detection process.

Future work could potentially involve quantization of LLMs thereby speeding learning and inference time. Researchers in their paper, DeepSyslog [52] separate event metadata such as timestamps from the rest of the log message. The text data is then preprocessed, converted into word embeddings, and processed into an LSTM model. Then the timestamps were concatenated with the outputs of the LSTM network by appending them to the feature vectors produced by the LSTM. These combined features (LSTM output and timestamps) are then fed into a dense (fully connected) layer for final classification. This process was designed to allow the LSTM to focus on learning the structure and patterns within the text data, while still preserving the time-based context provided by the timestamps. Future work on this would be beneficial in maximizing performance as timestamps are important fields to debug the system errors. Additionally, the model could be further developed to predict future events based on preceding log lines, providing valuable foresight for anomaly detection.

References

[1] The Business Research Company. Log management global market report. https://www.thebusinessresearchcompany.com/report/log-management-global-market-report, 2023. Accessed: 25.03.2023.

[2] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. pages 1285–1298, 10 2017. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134015.

[3] Xu Zhang, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, Dongmei Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, and Qian Cheng. Robust log-based anomaly detection on unstable log data. pages 807–817, 08 2019. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338931.

[4] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *International Joint Conference on Artificial Intelligence*, 2019.

[5] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. Deep learning for anomaly detection in log data: A survey. *Machine Learning with Applications*, 12:100470, 2023. ISSN 2666-8270. doi: https://doi.org/10.1016/j.mlwa.2023.100470. URL https://www.sciencedirect.com/science/article/pii/S2666827023000233.

[6] Dileepa Jayathilake. Towards structured log analysis. pages 259–264, 05 2012. ISBN 978-1-4673-1920-1. doi: 10.1109/JCSSE.2012.6261962.

[7] Elastic. Elastic stack. https://www.elastic.co/elastic-stack/, 2023. Accessed: 27.03.2023.

[8] S. Peltomaa. Elasticsearch-based data management proof of concept for continuous integration. Master's thesis, University of Oulu, Degree Programme in Computer Science and Engineering, 2022.

[9] Elastic. The history of elasticsearch. https://www.elastic.co/about/history-of-elasticsearch, 2023. Accessed: 04.02.2023.

[10] Elastic. Install elasticsearch, 2023. URL https://www.elastic.co/guide/en/kibana/8.3/install.html. [Online; 27.05.2023].

[11] Elastic. Configuring ssl, tls, and https to secure elasticsearch, kibana, beats, and logstash, 2019. URL https://www.elastic.co/blog/configuring-ssl-tls-and-https-to-secure-elasticsearch-kibana-beats-and-logstash. [Online; accessed 7-July-2023].

[12] Elastic. Elastic docs: Fleet and elastic agent overview. https://www.elastic.co/guide/en/fleet/current/fleet-overview.html. Accessed: 05.03.2023.

[13] Elastic Docs. Ingest Pipeines. https://www.elastic.co/guide/en/elasticsearch/reference/current/ingest.html. Accessed: 2023-07-04.

[14] Elastic. Install Fleet-managed Elastic Agent, 2023. URL https://www.elastic.co/guide/en/fleet/current/install-fleet-managed-elastic-agent.html. [Online; accessed 6-July-2023].

[15] Elastic. Install Fleet-managed Elastic Agent on Air-gapped environments, 2023. URL https://www.elastic.co/guide/en/fleet/8.3/air-gapped.html. [Online; accessed 6-July-2023].

[16] Elastic Docs. Kibana Query Langauge. https://www.elastic.co/guide/en/kibana/current/kuery-query.html, 2023. Online: Accessed: 2023-07-05.

[17] Jarkko Lagus. *Transformations and document similarities in word embedding spaces*. Phd thesis, University of Helsinki, Helsinki, Finland, June 2023.

[18] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[19] Bhuwan Dhingra, Hanxiao Liu, Ruslan Salakhutdinov, and William W. Cohen. A comparative study of word embeddings for reading comprehension, 2017.

[20] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

[21] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, 2012. doi: 10.1109/ICASSP.2012.6289079.

[22] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3973–3983. Association for Computational Linguistics, 2019. URL https://www.aclweb.org/anthology/D19-1410.pdf.

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

[24] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014. doi: 10.3115/v1/D14-1162.

[25] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, 2017.

[26] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[27] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. 04 2017.

[28] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. 02 2018.

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018. URL https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[32] E. Elbasani and D. Kim. Llad: Life-log anomaly detection based on recurrent neural network lstm. *Journal of Healthcare Engineering*, 2021, 2020. doi: 10.1155/2021/8829403. URL https://doi.org/10.1155/2021/8829403.

[33] Dwaipayan Biswas, et al. CorNET: Deep Learning Framework for PPG-Based Heart Rate Estimation and Biometric Identification in Ambulant Environment. *IEEE Transactions on Biomedical Circuits and Systems*, 13(2), 2019.

[34] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Lukasz and Polosukhin, Illia. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[35] ProjectPro. Understanding transformer architecture, 2023. URL https://www.projectpro.io/article/transformers-architecture/840. Accessed: 16.02.2023.

[36] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks, 2020.

[37] Yasaman Boreshban, Seyed Morteza Mirbostani, Gholamreza Ghassem-Sani, Seyed Abolghasem Mirroshandel, and Shahin Amiriparian. Improving question answering performance using knowledge distillation and active learning, 2021.

[38] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2006.

[39] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NeurIPS)*, 2015.

[40] XUECONG LIU. Distilling multilingual transformer models for efficient document retrieval: Distilling multi-transformer models with distillation losses involving multi-transformer interactions. Master's thesis, KTH Royal Institute of Technology, 2022.

[41] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[42] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding, 2020.

[43] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices, 2020.

[44] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[46] Radim Rehurek and Petr Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[47] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[48] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[49] François Chollet et al. Keras. `https://keras.io`, 2015.

[50] Anaconda Software Distribution. Anaconda software distribution. `https://anaconda.com`, Nov 2016. Accessed: [your access date here].

[51] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006. ISBN 978-0387310732.

[52] Junwei Zhou, Yijia Qian, Qingtian Zou, Peng Liu, and Jianwen Xiang. Deepsyslog: Deep anomaly detection on syslog using sentence embedding and metadata. *IEEE Transactions on Information Forensics and Security*, 17:3051–3061, 2022. doi: 10.1109/TIFS.2022.3201379.