FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Model-Agnostic Framework for Log Sequence Classification Applied to Anomaly Detection Using Fine-Tuned LLMs

João Rola Reis

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Jorge Barbosa

July 20, 2025

A Model-Agnostic Framework for Log Sequence Classification Applied to Anomaly Detection Using Fine-Tuned LLMs

João Rola Reis

Mestrado em Engenharia Informática e Computação

July 20, 2025

Abstract

As modern enterprises increasingly rely on complex IT infrastructures, the volume and diversity of log data generated from these systems have grown significantly, posing serious challenges to operational monitoring and anomaly detection. Effective log analysis is essential for maintaining system reliability, security, and efficiency, particularly in large organizations such as Grupo Salvador Caetano, SGPS S.A., which have increasingly larger systems to deal with the massive growth of the business expanding to multiple countries and sectors.

Recent advances in the area of deep learning, especially the development of large language models (LLMs), offer new opportunities for analyzing unstructured log data by learning complex sequential patterns. This thesis proposes a model-agnostic framework that leverages LLMs for log anomaly detection. The framework enables seamless integration of various LLMs, making it possible to experiment with and select the model best suited for a given dataset. Models are fine-tuned on normal log sequences, and anomalies are detected using a Top-K approach, flagging entries as anomalous that deviate from expected patterns.

The proposed framework is evaluated on three widely-used benchmark datasets, HDFS, BGL, and Thunderbird, where it consistently achieves competitive results, outperforming state-of-the-art methods in multiple scenarios. These results highlight the effectiveness of LLM-based log analysis and the importance of flexibility when selecting models based on operational needs.

Resumo

À medida que as empresas dependem cada vez mais de infra-estruturas de IT complexas, o volume e a diversidade dos logs gerados por estes sistemas aumentaram significativamente, colocando sérios desafios à monitorização operacional e à deteção de anomalias. A análise eficaz de logs é essencial para manter a fiabilidade, segurança e eficiência do sistema, particularmente em grandes organizações como o Grupo Salvador Caetano, SGPS S.A., que têm sistemas cada vez maiores para lidar com o crescimento massivo do negócio que se expande para vários países e sectores.

Os recentes avanços na área da deep learning, especialmente o desenvolvimento de LLMs, oferecem novas oportunidades para analisar logs não estruturados através da aprendizagem de padrões sequenciais complexos. Esta tese propõe uma solução agnóstica ao modelo que aproveita as LLMs para a deteção de anomalias. A solução permite a integração de várias LLMs, tornando possível experimentar e selecionar o modelo mais adequado para um determinado conjunto de dados. Os modelos são treinados em sequências normais de logs e as anomalias são detectadas utilizando uma abordagem Top-K, assinalando como anómalas as entradas que se desviam dos padrões esperados.

A estrutura proposta é avaliada em três conjuntos de dados de referência amplamente utilizados, HDFS, BGL e Thunderbird, onde obtém consistentemente resultados competitivos, superando os métodos do estado da arte em vários cenários. Estes resultados destacam a eficácia da análise de logs utilizando LLMs e a importância da flexibilidade na seleção de modelos com base nas necessidades operacionais.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) serve as a worldwide roadmap for creating a more sustainable and equitable future for everyone. The present work aligns with and contributes to the achievement of the following SDGs:

SDG 8 – Decent Work and Economic Growth: By improving the efficiency of log analysis and observability systems, this work supports more productive and resilient IT operations. Efficient monitoring and issue detection enable faster recovery and better resource management, contributing to sustained economic growth and increased operational effectiveness.

SDG 9 – Industry, Innovation and Infrastructure: The proposed solutions aim to modernize and optimize IT infrastructure through innovation in log analysis. These technological improvements foster more resilient and sustainable digital infrastructures, encouraging innovation and supporting long-term industrial development.

Acknowledgements

First and foremost, I would like to thank my parents for always being there for me and supporting me every step of the way. To my siblings, for being my company during every break throughout these years. And to my two grandmothers, who so often took care of me as if I were their own son.

I would like to express my sincere gratitude to my supervisor, Professor Jorge Barbosa, for taking a chance on both this thesis and on me, for providing guidance throughout the entire process, and for setting ambitious goals that pushed me to grow. I also want to thank the entire team at Grupo Salvador Caetano, especially the architecture team, Filipe Teixeira, and José Oliveira, for believing in me and supporting me whenever I needed help.

Last but not least, I want to thank my friends, in particular, my partners in most classes and projects, Afonso Baldo, Pedro Gomes, and Rui Pires, who made the journey much easier and from whom I learned so much.

João

"If you're afraid to fail, then you're probably going to fail."

Kobe Bryant

Contents

List of Figures

List of Tables

Listings

Abbreviations and Symbols

LLM	Large Language Mode
RAG	Retrieval-Augmented Generation
LCS	Longest Common Subsequence
PCA	Principal Component Analysis
SVM	Support Vector Machines
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
HDFS	Hadoop Distributed File System
BGL	BlueGene/L
LoRA	Low-Rank Adaptation

Chapter 1

Introduction

1.1 Context and Motivation

In today's increasingly digital world, observability has become a fundamental aspect of modern system architecture. As software systems grow in complexity, ensuring their reliability, security, and performance has become a critical challenge for organizations. Observability plays a key role in addressing this challenge by providing deep and actionable insights into the internal state of a system, enabling developers and system maintainers to detect, understand, and resolve issues more efficiently. By improving observability, teams are better equipped to respond to incidents in a timely manner, reduce system downtime, and enhance the overall user experience. Moreover, observability facilitates the identification of performance bottlenecks, security incidents, and anomalous user behavior, all of which are essential for maintaining the integrity and reliability of modern applications.

At the core of observability are three foundational pillars: metrics, traces, and logs [17]. These three data types, when used together, form a comprehensive framework for monitoring and understanding complex systems. Among these pillars, logs have experienced particularly significant growth in recent years. The sheer volume and complexity of log data generated by modern applications and infrastructure components have increased at an exponential rate. This rapid growth presents substantial challenges for organizations seeking to derive value from their logs. The traditional methods of log management, relying on manual analysis or simple keyword searches, are no longer sufficient to cope with the scale and diversity of today's log data. As a result, there is a growing need for advanced automatic techniques capable of efficiently storing, processing, and analyzing massive volumes of log data in real time. These techniques must not only be scalable but also intelligent, enabling systems to extract meaningful insights and detect anomalies automatically.

This challenge is particularly pressing for Grupo Salvador Caetano, SGPS S.A., a leading automotive and mobility solutions provider operating across various business domains and technological environments. The company generates and manages vast amounts of log data from its diverse

systems, spanning software applications, network infrastructure, customer-facing platforms, and internal business services.

GRUPO SALVADOR CAETANO, SGPS S.A. is a leading player in the sale and servicing of vehicles. Founded in 1946, the company has expanded beyond Portugal, establishing operations in several countries and representing several automotive brands. Business areas include car distribution and retail, vehicle manufacturing and assembly, transport services, car rentals, and workshops. As a result, a diverse array of log data is generated from multiple sources across their factories, car dealerships, and service workshops.

Given the scale of the company, it is crucial to have a robust system for log analysis and response. This capability allows Salvador Caetano to gain deeper insight into system behavior, enabling both proactive measures to maintain operational efficiency and faster, more effective reactive responses to quickly resolve any issues.

In recent years, the emergence of Large Language Models (LLMs) has opened new possibilities for analyzing unstructured textual data, which could be applied to logs. Leveraging LLMs for log analysis could be beneficial to identify complex patterns and generate valuable insights.

1.2 Problem Characterization

Large enterprises like Salvador Caetano often face significant challenges in analyzing and responding to system logs effectively, which could lead to extended problem resolution times. Additionally, as data volume continues to grow, the system may struggle to handle this scale, impacting response times and potentially compromising operational efficiency and reliability. This scaling problem is common in large enterprises, often requiring sophisticated clustering and analysis techniques to manage effectively [15].

Effective log analysis is crucial for a company like Salvador Caetano, which has extensive and complex systems. Timely insights into system behavior can prevent operational disruptions and enable proactive decision-making [22]. Moreover, improved log analysis offers benefits beyond operational efficiency, enhancing security, ensuring compliance, and providing valuable business insights.

A core challenge in log analysis lies in the nature of the logs themselves. These logs are typically unstructured and are written manually by developers during the development process. While they are often understandable to humans, they can be extremely difficult for machines to interpret and extract meaningful insights from. Even within the same system, the format and structure of logs may vary significantly, making it even harder to automate their analysis. This problem becomes even more noticeable in distributed environments, particularly with the widespread adoption of microservices, where the complexity and variability of log data increases.

To address these challenges, various approaches could be explored. Natural Language Processing techniques could potentially be applied to extract meaningful information from unstructured log messages. Additionally, advanced algorithms, including traditional statistical methods, might be employed to detect anomalies and automate the classification of log entries. This thesis

proposes the development of a log analysis framework that leverages the capabilities of LLMs to process and interpret unstructured log data, aiming to identify an effective strategy for improving log analysis capabilities in large-scale enterprises like Salvador Caetano. By designing the framework to be adaptable and model agnostic, this work also aims to future-proof log analysis workflows, allowing organizations to integrate emerging LLM technologies without needing to redesign their entire observability pipeline.

1.3 Contributions

Within the broader scope of observability in large-scale systems, this thesis specifically focuses on the domain of log analysis. The primary objective is to contribute to the advancement of current log analysis methodologies by examining the limitations of existing approaches and proposing solutions to overcome them. To address these challenges, this thesis introduces a model-agnostic log analysis framework, designed with flexibility in mind. The framework is structured such that the underlying large language model (LLM) can be easily replaced or updated without affecting the functionality of the overall system.

To validate the effectiveness of the proposed framework, it will be implemented and evaluated using multiple benchmark datasets that are widely recognized in the field of log analysis and anomaly detection.

1.4 Document Structure

This chapter establishes the starting point and primary focus of this thesis, outlining the context in which the work is being developed, its main objectives, and the proposed solution. The remainder of this document is organized as follows:

1. Chapter 2, **Observability on Large Systems:**

   This chapter introduces key concepts and definitions that are essential for understanding the rest of the thesis. It presents the current state of the art in log parsing and log analysis, and concludes with an overview of existing market solutions for log management.

2. Chapter 3, **A Framework for Log Analysis:**

   This chapter provides a detailed description of the proposed framework, including its architecture and implementation. It highlights how the framework is developed so it's model agnostic.

3. Chapter 4, **Experimental Work:**

   This chapter presents the experimental evaluation of the proposed solution. The framework is tested against state-of-the-art approaches using three widely adopted benchmark datasets for log anomaly detection.

4. Chapter 5, **Conclusions:**

   This chapter summarizes the work conducted throughout the thesis, reflects on the main findings, and outlines potential directions for future research and improvement.

Chapter 2

Observability on Large Systems

This chapter starts by presenting the fundamental concepts about observability in large systems, then explores the different ways to train large language models (LLMs), goes through the state-of-the-art in log parsing strategies, log analysis techniques, and finally, tools for log management.

2.1 Observability

Observability is the ability to understand a system's internal state by analyzing its external outputs and interactions, without the need for intrusive instrumentation [17]. The primary goal is to detect and resolve performance or security issues, enhance reliability, and gain visibility into complex systems to identify failures and bottlenecks.

2.1.1 The Three Pillars of Observability

The foundation of modern observability systems is built on three core types of telemetry data: metrics, traces, and logs [1]. Together, these data types provide a comprehensive view of system behavior and performance.

1. **Metrics:** Quantitative measurements of system performance and resource usage over time. They offer a high-level overview of system health. Examples include CPU usage and memory consumption.

2. **Traces:** Records that follow the flow of a request through the system, from the frontend to the backend and across internal service calls. Traces help identify the path and performance of specific requests, making it easier to locate bottlenecks. An example is a trace showing how a user request passes through different microservices.

3. **Logs:** Detailed records of events that occur within the system. Logs are essential for debugging and provide contextual information about specific system behaviors. Examples include error messages or debugging messages.

2.2 Training of LLMs

There are different ways of injecting knowledge into LLMs. Although LLMs encapsulate an enormous quantity of factual information, this knowledge may not be sufficient or specific enough for the problem at hand, in this case, log analysis. There are different ways we can improve the ability of LLMs to succeed in a specific need, and these will be explored in this section.

2.2.1 Prompt Engineering

Prompt engineering [2] is the practice of crafting high-quality prompts to better guide large language models (LLMs) toward generating the desired outputs. This process takes into account several factors, such as prompt length, writing style, and structure, aiming to determine the most effective way to present the input for the specific task.

In the context of anomaly detection in log sequences, an example of prompt engineering could involve providing the log sequence to be analyzed, along with examples of normal patterns and known rules that typical log sequences should follow. This additional context helps the LLM better understand what constitutes normal behavior and improves its ability to identify anomalies.

2.2.2 RAG

Retrieval-Augmented Generation (RAG) [14] is a technique that improves the capabilities of large language models (LLMs) by including external knowledge sources. This can be done without additional training, simply by enabling the model to access and retrieve relevant data during inference. The core idea is to take an input query and retrieve documents from a knowledge base that are similar to it.

In the context of anomaly detection in log sequences, RAG could be used to retrieve known anomalous patterns or sequences from a database. These retrieved examples could then be provided to the model, allowing it to leverage this additional context and improve its analysis and anomaly detection performance.

2.2.3 Fine-tuning

Fine-tuning is the process of adapting a pre-trained model using a smaller, task-specific dataset to improve its performance on a particular objective. According to [24], there are three main types of fine-tuning: supervised fine-tuning, reinforcement learning, and unsupervised fine-tuning.

In supervised fine-tuning, the model is trained on pairs of labeled input and output data, where the input typically consists of an instruction and the output represents the desired response. In reinforcement learning, the model generates a response and receives feedback in the form of rewards or penalties, depending on the quality of the response. Finally, unsupervised fine-tuning continues the training of the original language model in a causal, autoregressive manner; in other words, it trains the model to predict the next token of the sequences from the new, smaller dataset.

As seen in the solutions discussed in Sections 2.4.1.5 and 2.4.1.6, a combination of reinforcement learning and unsupervised fine-tuning is typically employed, where the small dataset is composed of only normal log sequences so the underlying model can learn the patterns of these sequences.

2.2.3.1 Low-Rank Adaptation

As large language models continue to grow in size, full fine-tuning, which involves updating all model parameters, becomes increasingly impractical due to the significant computational and memory demands. Low-Rank Adaptation (LoRA) [12] is an improved fine-tuning technique designed to address this issue by drastically reducing the number of trainable parameters.

Instead of updating the original weight matrices in the model, LoRA introduces a pair of small, trainable, low-rank matrices that approximate the update. These low-rank matrices are the only parameters trained during fine-tuning, while the original model weights remain frozen.

This approach maintains model performance while significantly reducing training time and memory usage. Furthermore, because the low-rank matrices can be merged with the original weights after training, LoRA does not introduce any additional inference latency, unlike many other parameter-efficient tuning methods.

2.2.3.2 Unsloth

Unsloth [3] is an open-source library designed to make fine-tuning large language models faster, lighter, and easier, without compromising accuracy. Training LLMs can be challenging, especially when switching between different models, but Unsloth simplifies this process by streamlining the entire training workflow, including model loading, training, and saving. It supports all major open-source LLMs, and new models are quickly integrated into the platform as they are released.

In some approaches, such as those described in Sections 2.4.1.5 and 2.4.1.6, the solutions are tightly coupled to the architecture of the underlying model. Unsloth addresses this limitation by providing a unified interface for working with various LLMs, enabling the development of model-agnostic solutions for log analysis.

2.3 Log Parsing

Log parsing involves transforming unstructured log data into a structured format that can be efficiently analyzed using various log analysis techniques [36]. Unstructured logs are typically human-readable strings containing natural language text mixed with variable values. Although these logs are easily interpretable by humans, their lack of a consistent format makes them difficult for machines to process.

In contrast, structured logs are formatted objects that machines can understand. This structured format facilitates subsequent analysis by enabling automated log analysis algorithms to extract meaningful insights. Without this structure, log analysis would be extremely difficult, as raw,

unstructured data lacks the organization necessary for the machine to process, making it almost impossible to obtain insights.

It's often tough to transform raw, unstructured logs into a structured format. This difficulty stems from clearly identifying variables versus constants within the log entries. The constant portion is what defines a log template, remaining consistent across all instances of a specific event. Meanwhile, the variable portion contains dynamic data that will be unique to each occurrence of that event. The main objective of log parsing is to distill each raw log into its specific log template.

The importance of log parsing becomes even more apparent in the context of anomaly detection. Since logs record detailed execution information, they serve as a valuable resource for identifying system anomalies. Current research in this field often leverages machine learning techniques, for which log parsing is a crucial preprocessing step in training the models [38].

2.3.1 Classification of Log Parsing Methods

According to [38], parsing methods can be categorized into three types of log parsers: frequent pattern mining, clustering, and heuristics. In addition, [36] introduces an additional category called iterative partitioning.

Frequent pattern mining is based on the idea that log templates can be constructed from constant tokens that frequently appear in logs. This method is relatively straightforward: the parser scans the dataset multiple times to identify tokens that occur repeatedly, forming tuples of token positions and values to build frequent itemsets. Using these itemsets, the parser clusters similar logs and generates corresponding log templates.

Clustering offers an alternative approach, where logs are grouped into clusters based on similarity, and common tokens within each cluster are extracted to form log templates. Unlike frequent pattern mining, which processes the entire dataset to find recurring patterns, clustering focuses on grouping similar log messages and identifying shared components.

Heuristics represent another strategy, using predefined rules and expert knowledge to guide the parsing process. This method leverages domain-specific insights to improve both the efficiency and accuracy of the parser by separating log messages according to logical or structural cues.

Finally, iterative partitioning involves progressively refining the grouping of logs in each iteration based on predefined rules, aiming to converge toward an optimal solution over time.

2.3.2 Offline Log Parsers

Offline log parsers operate on pre-collected log data [38]. This approach requires the entire dataset to be available before processing, making it particularly well-suited for scenarios where logs are analyzed periodically rather than in real time.

In practice, offline log parsing is commonly used in batch analytics and historical trend analysis, where the emphasis is on depth of insight rather than immediacy.

2.3.2.1 IPLoM

One example of this type of log parser is the Iterative Partitioning Log Mining (IPLoM) algorithm [18]. IPLoM employs a structured four-step process to parse logs. Initially, it groups log messages by length, assuming similar-sized logs share structural features. It then identifies token positions with the fewest unique values within each group, splitting logs further based on these values to isolate consistent patterns. In the final partitioning step, IPLoM identifies a one-to-one mapping, or bijective relationship, between unique tokens in specific token positions to differentiate variable tokens from constant structural elements. By determining whether token positions are constant or variable within each partition, IPLoM generates concise line formats that represent the log message structure.

2.3.2.2 LogMine

Another example of this type of algorithm is LogMine [9]. The algorithm begins with tokenization and the use of domain-specific regular expressions to normalize the data. An adapted version of the UPGMA algorithm [28] is used for clustering log messages, grouping similar messages into clusters based on their distances.

2.3.3 Online Log Parsers

Online Log Parsers parse logs dynamically as they are received [38]. This capability enables real-time analysis, which makes it particularly well-suited for scenarios where immediate insights are critical, such as system monitoring and anomaly detection.

In practice, online log parsing is essential for applications that require real-time observability, including infrastructure monitoring platforms, continuous integration/continuous deployment (CI/CD) systems, and automated incident response frameworks.

2.3.3.1 Spell

One example of this type of log parser is the Spell algorithm [4]. This algorithm uses the Longest Common Subsequence (LCS) approach to dynamically extract patterns from un- structured logs. By comparing new log messages with existing patterns, Spell determines if they fit an existing message type or necessitate the creation of a new one. It utilizes data structures like LCSObjects, which hold parsed LCS sequences and related metadata, organized within an LCSMap. Logs are tokenized and matched against existing patterns using these structures. If a match is found, the metadata is updated, otherwise, a new LCSObject is created. To improve efficiency, Spell incorporates a pre-filtering strategy using a prefix tree, which reduces the computational overhead of comparisons by hierarchically organizing message type prefixes, avoiding the need to compare the LCS with all existing message types. However, this approach may occasionally misclassify logs when overlapping message structures result in incorrect matches.

2.3.3.2 Drain Parser

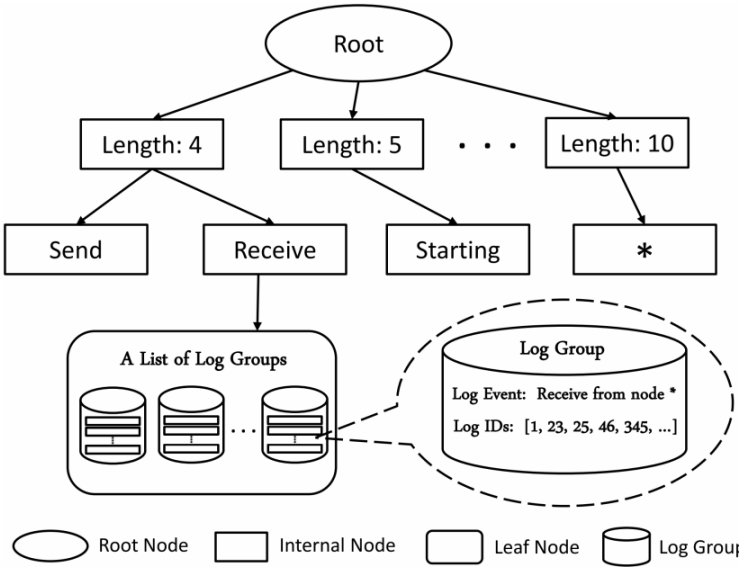

Figure 2.1: Architecture of the Drain parser, reprinted from [11]

Among all log parsers, Drain [11] is by far the most widely used. In fact, nearly all major log analysis papers rely on this parser due to its high efficiency and accuracy.

Drain organizes parsed logs into log groups, where each group consists of:

- log event, which is a generalized log template representing similar messages
- a list of log IDs, corresponding to individual log messages that match the event.

The parsing process begins with a lightweight preprocessing step, which applies domain-specific regular expressions to remove variable tokens, such as IP addresses, from log messages. Most datasets only require a small number of such expressions, making this step lightweight.

After preprocessing, Drain uses a fixed-depth tree to group messages efficiently. This tree structure is guided by two main hyperparameters: `depth`, which limits the depth of the tree, and `maxChild`, which limits the number of children per node. These constraints prevent the tree from becoming unbalanced or too large, ensuring fast parsing.

In the first level of the tree, logs are grouped based on message length, under the assumption that log messages from the same event usually have the same number of tokens. Deeper levels of the tree branch based on the values of tokens at the beginning of the log messages, since these are more likely to be constants across similar logs. To reduce branching caused by dynamic values, tokens that contain digits are ignored during traversal.

When a message reaches a leaf node in the tree, it is compared against the existing log events within that node. The similarity between the new log message and each existing log event is computed as:

$$\text{Similarity} = \frac{\sum_{i=1}^{n} \text{seq}_1[i] == \text{seq}_2[i]}{n} \tag{2.1}$$

Here, `seq_1` is the sequence of tokens in the log event, and `seq_2` is the sequence of tokens in the current log message.

If the similarity score exceeds a predefined threshold, the message is added to the corresponding log group. Additionally, the log event is updated, where for any position where the token differs between the log message and the event, that token in the event is replaced by a wildcard. This ensures the template remains general enough to accommodate variation.

If no existing log event exceeds the similarity threshold, a new log group is created. The log message itself becomes the initial log event, and the group is initialized with the current message's ID.

2.4 Log Analysis

Log analysis involves extracting insights and identifying patterns from log messages. This can be achieved using traditional statistical methods or by leveraging machine learning techniques. The primary goal is to monitor the system, which allows it to detect and respond to potential issues, consequently improving its reliability and efficiency.

Historically, insights were extracted from logs using keyword-based methods, relying on terms such as "error," "info," and "warning." However, with the exponential growth of log data, such approaches have become inefficient and error-prone. This has led to the development of more advanced methods that can automatically extract meaningful insights from unstructured log messages.

2.4.1 Anomaly Detection in Log Sequences

Anomaly detection in log sequences focuses on identifying deviations from established patterns, which may indicate software failures, security breaches, or other operational issues. Therefore, it is crucial for the detection algorithm to understand the complex relationships between log events from a log sequence, so as to recognize unusual behavior accurately. A log sequence is considered anomalous if it differs from the expected normal flow of the program.

In this section, an overview of traditional machine learning approaches will be explored. Following this, methods that leverage Long Short-Term Memory (LSTM) will be investigated, and finally, methods employing the Transformer architecture will be examined. This includes solutions utilizing Large Language Models (LLMs), which will serve as the primary point of comparison with the framework proposed in this thesis.

2.4.1.1 Traditional Machine Learning Methods

Several traditional machine learning algorithms have been successfully applied to log analysis.

One example is presented in [34], where log messages are first parsed and structured, and then feature vectors are constructed. Principal Component Analysis (PCA) [6] is applied to these vectors to detect anomalies by identifying patterns that deviate from the norm.

Another approach is described in [16], where the Isolation Forest (iForest) algorithm is used. In the training phase, the algorithm builds multiple isolation trees by recursively partitioning random subsets of the data. Each data point (log message) is isolated by randomly selecting features and split points. In the evaluation phase, an anomaly score is computed for each instance based on the average path length across the trees, with shorter paths indicating higher anomaly scores, as anomalies are easier to isolate.

A notable method is proposed in [32], where One-Class Support Vector Machines (SVMs) are applied to anomaly detection in logs. This approach uses specialized kernel functions designed to capture the structural patterns within system logs, enabling effective identification of anomalous events.

2.4.1.2 DeepLog

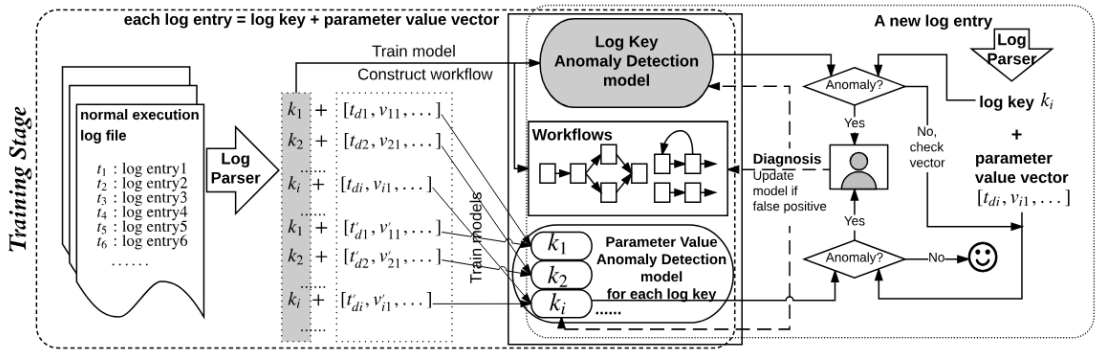

Figure 2.2: Architecture of the DeepLog framework, reprinted from [5]

With recent advances in neural networks, it's no surprise that log analysis has become a promising field for their application. For example, Long Short-Term Memory (LSTM) networks have been explored in two distinct approaches to log analysis.

One such approach is the DeepLog model [5], which reveals a pattern in the training and inference processes that is common across many deep learning-based models, including those leveraging large language models (LLMs).

In the DeepLog paper, logs are parsed using the Spell parser [4], rather than the more commonly used Drain parser [11]. This decision is likely due to the publication date of the paper, which predates the release of the Drain parser. After parsing, log keys are extracted and organized into log sequences. The method of grouping log entries into sequences varies depending on the dataset. DeepLog then introduces an innovative step, where it creates a vector for each log key containing both the original log parameters and the time elapsed since the previous log message. This means the model is trained not only on the parsed log key but also on the original parameters and time intervals. As a result, it can detect not only anomalies in the sequence of log keys but also unusual parameter values and unexpected delays between log messages.

The training phase is divided into two parts: one for detecting execution path anomalies, and the other for parameter value and performance anomalies. The first part represents the beginning of

a trend later adapted by many deep learning-based log analysis methods. In this step, the model is trained solely on logs generated during the normal operation of the system. For each log sequence, a sliding window of size h is created and passed through a series of LSTM layers. These layers are trained to predict the next log key based on a window composed of the previous h log keys. In essence, the model learns the relations and in what order the log keys should be in normal log sequences.

The second part of the training focuses on detecting anomalies in parameter values and performance. Each sequence of parameter vectors (including the parameters and the time elapsed between log keys) is treated as a multivariate time series. A similar LSTM-based approach is used, where the model is trained to minimize the difference between the predicted and actual parameter vectors.

During inference for execution path anomaly detection, a window of log keys is provided to the trained DeepLog model. The model outputs a probability distribution over the possible next log keys. Instead of checking whether the predicted log key exactly matches the next log key, which would be overly strict given that multiple normal outcomes may follow a similar history, DeepLog uses a top-g strategy. If the actual next log key is among the top g most probable candidates, the sequence is considered normal, otherwise, it is flagged as anomalous.

For parameter value anomaly detection during inference, the model computes the mean squared error (MSE) between the predicted and actual parameter vectors. Instead of relying on an arbitrary threshold, the training dataset is split into training and validation sets. The model is applied to the validation set to compute the distribution of MSE values, which is then modeled as a Gaussian distribution. If a new prediction's error falls within a high-confidence interval of this distribution, it is considered normal, otherwise, it is labeled as anomalous.

Two key parameters significantly impact the performance of the DeepLog method: the window size h and the top-g threshold for execution path anomaly detection. These parameters are selected as follows: h is set to the length of the shortest log sequence in the dataset, and g is set to the maximum number of branches observed at any divergence point in the dataset's log sequences.

Lastly, this solution includes an important and innovative feature, which is the ability to update the model online. When the model incorrectly flags a normal log as an anomaly, the user can provide feedback, prompting the model to adjust its weights accordingly. This enables the model to adapt over time to changes in system behavior.

2.4.1.3 LogAnomaly

Another great example of using LSTMs is LogAnomaly [19]. It follows a different process and, according to the paper, achieves better results than DeepLog.

After log parsing and template creation, LogAnomaly enters an offline learning phase. This phase is trained exclusively on normal log sequences, captured during the system's normal operation. Inspired by the popular Word2Vec model [20], it introduces Template2Vec, which represents log templates by placing extra emphasis on synonymy and antonymy.

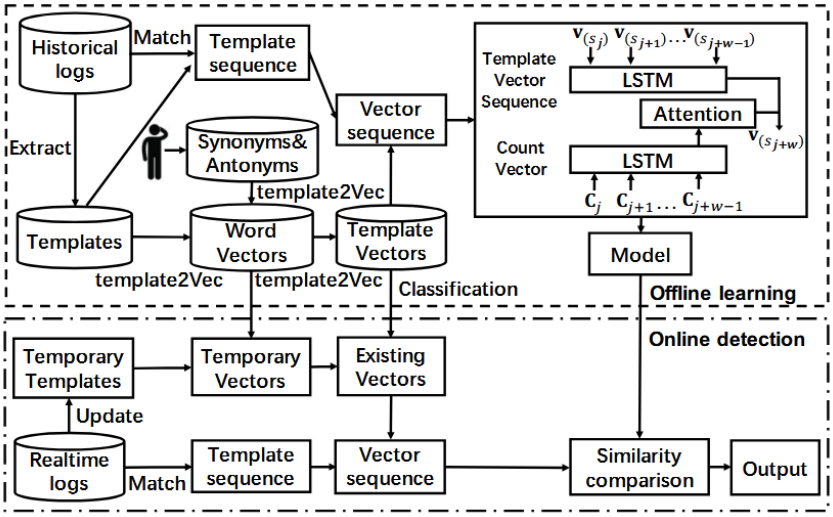

Figure 2.3: Architecture of the LogAnomaly framework, reprinted from [19]

First, a set of synonyms and antonyms must be constructed. While there are existing lexical databases containing English synonyms and antonyms, domain experts may need to update or extend this list to include domain-specific terms. These are then used to generate word vectors with dLCE [21], which incorporates this semantic information to ensure that synonyms and antonyms are appropriately represented.

From the resulting word vectors, template vectors are computed as the weighted average of the word vectors comprising each log template. These template vectors are fed into an LSTM neural network, which is trained to learn both sequential patterns (the typical order in which log events occur) and quantitative patterns (the relationships between the frequencies of different log events within a given time window, for example, certain events may always happen the same amount of times).

During the log anomaly detection phase, each incoming log is matched to an existing template and converted into the corresponding template vector. This vector is then fed into the trained LSTM, which predicts the next expected log event. If the actual next log event is among the top-k predicted by the LSTM, the sequence is considered normal, otherwise, it is flagged as anomalous. Unfortunately, it is not clear how the k is defined.

LogAnomaly also has a novel approach to handling previously unseen log types. It generates a temporary template vector for the new log and computes its similarity to existing templates. The new log is then approximated by mapping it to the most similar known template.

What distinguishes LogAnomaly from other methods is its unique way of representing log templates and log sequences as vectors, capturing both semantic and sequential relationships.

2.4.1.4 LogBERT

In recent years, the Transformer architecture [31] has completely revolutionized the field of computer science. It is no surprise that this transformative technology has also been explored and

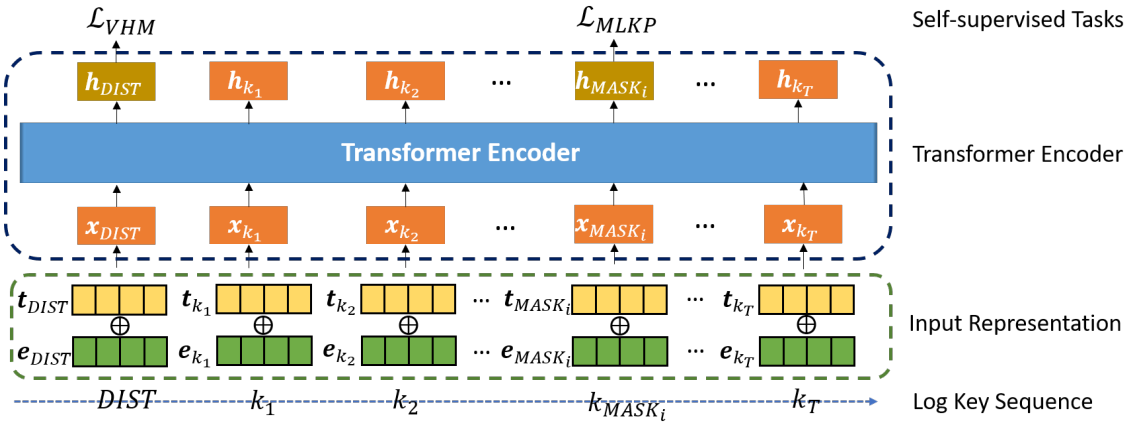

Figure 2.4: Architecture of the LogBERT framework, reprinted from [8]

applied in the domain of log analysis.

One of the first approaches to adopt this paradigm is LogBERT [8]. The framework starts by using the Drain parser [11] to parse raw logs and generate log keys. The training phase then begins, using only normal log sequences.

These sequences are modeled using two types of information: log key embeddings and position embeddings. Each log key is mapped to a dense vector in the log key embeddings. Position embeddings are used to inform the model of the position of each log key within the sequence, a crucial aspect, since the order of events can be highly significant. A sinusoidal function is employed to generate a unique vector for each position in the sequence.

LogBERT in the training phase leverages a Transformer encoder to learn relationships among log keys in normal sequences based on two tasks:

1. **Masked Log Key Prediction:** A portion of the log keys in a sequence is randomly masked, and the model is trained to predict the masked keys. The objective is to accurately reconstruct the missing log keys. The cross-entropy loss function is used here, penalizing incorrect predictions.

2. **Volume of Hypersphere Minimization:** This task is based on the assumption that normal log sequences should be clustered closely together in the embedding space, while abnormal ones should lie further away. To achieve this, a center point is calculated by averaging all the vectors from the normal log sequences. A loss function is then used to compute the average squared distance of each sequence to this center. Minimizing this loss forces the model to represent all normal logs close to the center, thus improving its ability to distinguish anomalies and predict masked tokens with higher accuracy, since normal sequences share similar patterns.

The final training objective combines the loss functions from both tasks using a weighted sum.

After training, the inference (anomaly detection) phase begins. Given a log sequence, tokens are randomly masked, and the model predicts the top-g most likely log keys for each masked

token. Then, if more than r log keys in the sequence are not among the top-g predicted candidates, the sequence is flagged as anomalous. Otherwise, it is considered normal. The hyperparameters r and g are tuned using a small validation dataset.

This solution marks an important shift in the field of log analysis, introducing the use of Transformer-based architectures to model and detect patterns in system logs.

2.4.1.5 LogGPT

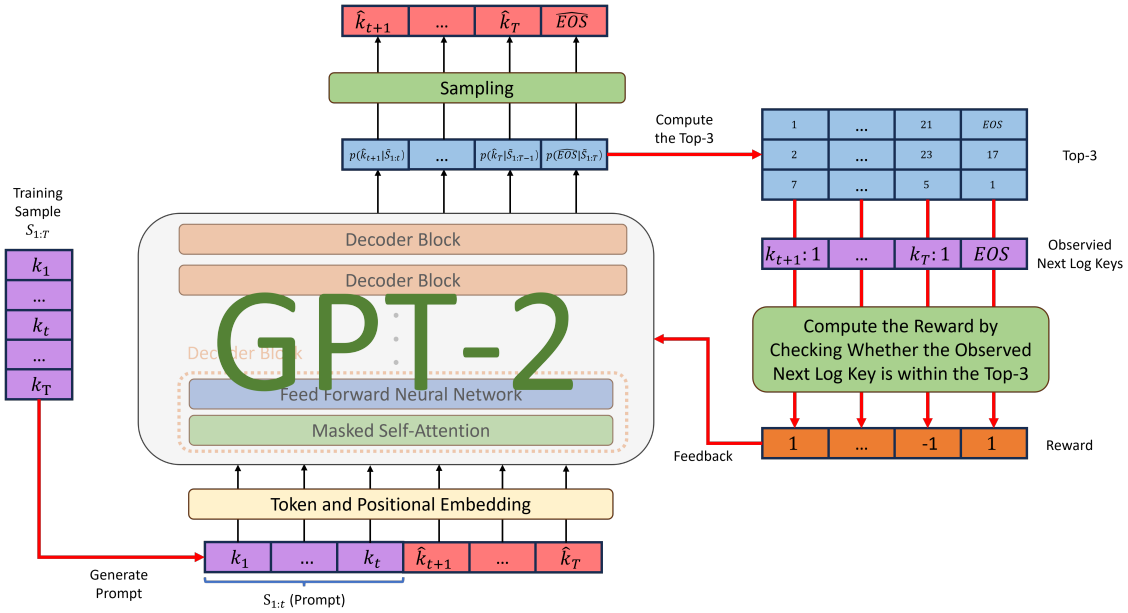

Figure 2.5: Architecture of the LogGPT framework, reprinted from [10]

More recently, with the release of GPT, large language models (LLMs) have emerged across various fields of computer science. Log analysis is no exception, and LLMs are beginning to be applied in this domain as well.

One of the first approaches to explore this type of solution is LogGPT[10], which leverages GPT-2[26] as the underlying model for log analysis. LogGPT demonstrates impressive results when compared to all previously discussed methods. As is common, the Drain parser [11] is used to extract log templates from raw log messages, followed by a hash function to generate a unique identifier for each template. Specifically, MD5 is used, and only the first 8 hexadecimal characters are retained, providing a compact and consistent identifier for each log template.

With the log keys arranged as sequences, the training (or fine-tuning) phase begins. Only normal log sequences are used. LogGPT is trained to predict the next log key given the preceding keys in the sequence. The objective function used is the cross-entropy loss applied to all sequences in the training dataset. After training, the model can generate log sequences based on a partial input sequence.

Next, a reinforcement learning phase is introduced to improve the model's performance. The reward mechanism works as follows: at each step, if the actual log key is within the Top-K predicted log keys, the model receives a reward of 1, otherwise, it receives a reward of -1. The use of Top-K predictions (instead of only the top-1) is critical, as the relationships between log events can be complex, and multiple valid sequences may exist. Relying solely on the single most probable prediction could negatively affect performance in such cases.

During the inference (anomaly detection) phase, a log sequence is evaluated to determine whether it is anomalous. For each log key, excluding the first, since there is no prior data to base a prediction on, the model checks whether the actual log key appears in the Top-K predicted keys. If all log keys in the sequence fall within the Top-K predictions, the sequence is considered normal, otherwise, it is flagged as anomalous.

To define the value of K, the authors suggest using 50% of the total number of unique log keys in the training dataset. This means that a log key is considered acceptable if it falls within the top half of all possible log keys as predicted by the model.

This solution represents a significant milestone in the field of log analysis, as it is among the first to incorporate large language models into this domain.

2.4.1.6 LogLLaMA

Another notable approach is the LogLLaMA framework [37], which adopts a training strategy similar to that of LogGPT. Like LogGPT, it is trained to predict the next log key based on preceding keys and incorporates a reinforcement learning mechanism to further improve anomaly detection. During inference, LogLLaMA also relies on Top-K prediction, setting K to 50% of the total number of unique log keys in the training set.

The main distinctions between the two frameworks lie in the underlying model architecture and training data utilization. LogLLaMA uses LLaMA 2, although it does not specify which version it uses, instead of GPT-2 and follows a proportional training approach, using 10% of the dataset for training. In contrast, LogGPT trains on a fixed sequence of 5,000 log lines, regardless of the dataset size. This leads to LogLLaMA having significantly more data for training in the tested datasets.

However, it is important to note that no official code implementation for LogLLaMA is available at the time of writing. This lack of code makes it significantly more difficult to perform thorough analysis or to compare its performance under the same conditions as other published methods.

2.5 Tools for log management

A variety of solutions are available that streamline log collection, parsing, storage, analysis, and visualization. While this thesis primarily focuses on log analysis algorithms, it's important to understand how these tools can implement a real-world system pipeline that could then be integrated with our proposed framework.

2.5.1 ELK Stack

A example of this tools is the ELK Stack, a full stack logging service developed by Elastic™ [25]. It comprises three main open-source components: Elasticsearch, Logstash, and Kibana. In addition to these core tools, Elastic offers supplementary products such as Filebeat and Metricbeat that give the user extra functionality.

Elasticsearch is a distributed search engine that provides full-text search capabilities. It allows users to query data efficiently through a RESTful API. As part of the ELK Stack, Elasticsearch also functions as a datastore, where logs are stored in a format optimized for quick retrieval using Elasticsearch indices.

Logstash is a data processing pipeline that enables the ingestion of data from multiple sources, performs parsing and transformation, and then forwards the processed data to Elasticsearch for storage and analysis. A key feature of Logstash is its Grok plugin, which can parse unstructured logs into structured formats using personalized patterns. Additionally, Ruby scripts can be ran within Logstash to provide extra filtering and transformation capabilities.

Kibana is a visualization tool that enables users to analyze data stored in Elasticsearch. It provides a wide range of features for creating charts, graphs, and other types of visualizations, including dashboards that consolidate multiple views of data. The user interface is accessible through a web browser, making it easy for users to explore and monitor their data in real time.

Filebeat is a log shipper designed to collect and forward log data to a specific pipeline, such as Logstash. Users can configure which log files to monitor and forward. The process of collecting log data is called "harvesting," which involves reading individual log lines from the configured log files and sending them to the specified output.

Metricbeat is a data shipper designed to collect and forward system and application metrics, such as memory usage, CPU statistics, and other performance indicators. Similar to Filebeat, users can configure the desired output destination, such as Logstash.

An example workflow using the ELK Stack could proceed as follows:

- **Log Collection:** Filebeat collects logs configured by the user and forwards them to Logstash, while Metricbeat gathers key metrics about system performance, such as CPU usage and memory consumption, and sends them to the same pipeline.

- **Log Parsing and Transformation:** In Logstash, the Grok plugin is used to parse unstructured log data into structured formats. Additionally, a Ruby script can be applied to filter the data and generate new fields based on calculated values from the original log entries.

- **Data Storage:** The processed data is then stored in Elasticsearch. Using Elasticsearch indices, the data can be efficiently retrieved for analysis.

- **Data Visualization:** Finally, the data becomes accessible through Kibana's web-based interface, where users can create dashboards and analyze logs.

2.5.2 OpenTelementry

OpenTelemetry [27] is an open-source observability framework that standardizes the collection and management of telemetry data (logs, metrics, and traces) in distributed systems. By providing implementations across multiple programming languages and frameworks, it simplifies the integration of observability into modern applications. Its distributed tracing capabilities enable end-to-end visibility of request flows across service boundaries, making it particularly valuable for debugging and performance analysis in microservices architectures.

The OpenTelemetry Collector is a central component that receives, processes, and exports telemetry data from applications. It supports multiple input data formats and can simultaneously export to different backend systems. OpenTelemetry provides language-specific SDKs that implement the OpenTelemetry API, enabling developers to instrument applications in various programming languages. The collector's exporter components handle sending telemetry data to backend systems, making it easy to switch between different backends without changing application code. Importantly, OpenTelemetry itself doesn't store data; it's purely a collection and transmission framework. For data storage and visualization, a solution could be using Elasticsearch to store the telemetry data, which can then be visualized using Kibana, as explained in the last section.

2.5.3 Datadog

Datadog is a leading Software-as-a-Service (SaaS) observability platform [1], designed as a unified solution for collecting, analyzing, and visualizing telemetry data across the entire technology stack. Unlike the tools discussed so far, Datadog includes native real-time log analysis capabilities out of the box.

It offers significantly more features than any previously mentioned tools. Datadog provides built-in integrations with a wide range of applications, services, and systems, including support for OpenTelemetry. It uses an agent to collect and transmit data, and includes interactive dashboards for visualizing and manually analyzing log data, features also available in platforms like the ELK Stack. However, Datadog goes further by offering user behavior insights, allowing developers to make more informed, data-driven product decisions.

For the purposes of this thesis, the most relevant feature is Datadog Watchdog. This AI-powered functionality enables real-time log analysis by automatically generating alerts, identifying anomalies, and providing root cause analyses based on telemetry data collected across the platform. It can detect spikes, drops, and unusual patterns in key health indicators without requiring manual configuration. Watchdog is capable of identifying problematic deployments and potential system failures before they escalate.

Despite its powerful capabilities, Datadog is not without drawbacks. It is a premium service and can be very expensive, especially for smaller to medium-scale use cases. Another concern is vendor lock-in, a common issue with SaaS platforms. Applications may become tightly coupled with Datadog's ecosystem through custom metrics and proprietary agents, making it difficult and costly to migrate to another observability solution later on.

Chapter 3

A Framework for Log Analysis

This chapter begins by outlining the problem this work aims to address, discussing the limitations of existing literature, defining the intended objectives, and presenting the research questions and scope of the project. It then describes the proposed framework in detail, covering each stage of the process, from data preprocessing and model fine-tuning to the inference phase.

3.1 Literature's Limitation

The previous chapter examined the current state-of-the-art in log analysis, covering traditional machine learning methods, deep learning approaches based on long short-term memory (LSTM) neural networks, and, more recently, the use of large language models (LLMs) for this task.

An analysis of the results presented in the literature suggests that LLM-based approaches currently achieve the highest performance, positioning them at the forefront of research in this area. This highlights the growing trend toward adopting LLMs as the preferred direction for improving log analysis systems.

However, a significant limitation observed in these approaches is the strong coupling between the solution framework and the specific LLM architecture used. Both LogGPT [10] and LogLLaMA [37] exemplify this issue. LogGPT relies on GPT-2.5, and its architecture is tailored specifically to that model, while LogLLaMA appears similarly bounded to LLaMA 2. For LogLLaMA, although no code is provided, there is no information suggesting otherwise.

Given the rapid evolution of LLMs and the continuous emergence of newer, more powerful models, this lack of modularity poses a challenge. These frameworks do not provide the flexibility needed to easily swap out or upgrade the underlying model. As a result, they fall short in supporting experimentation and adaptation to newer models in this fast-moving field.

Another consequence of this tight integration is the inability to scale the solution down when needed. In certain real-world scenarios, the highest level of model complexity may not be necessary to achieve satisfactory performance. In such cases, a smaller and more efficient model could

provide similar results while significantly reducing computational and operational costs. The current lack of modularity prevents practitioners from easily replacing the model with a lighter alternative, thus limiting the ability to optimize for cost and efficiency.

3.2 Desired Achievements

To address the lack of modularity identified in the literature, this work sets out to achieve the following objectives:

1. Implement a flexible framework that allows users to easily switch between different underlying models.
2. Evaluate the performance of the framework using various models across multiple datasets, in order to assess the benefits of model interchangeability.

3.2.1 Scope

This work focuses on building and evaluating a modular log analysis framework based on large language models (LLMs), aiming to provide flexibility in selecting and switching between models. The scope includes:

- Implementing a framework that allows the training and evaluation of LLMs on log sequence anomaly detection tasks.
- Supporting multiple LLMs of different sizes and architectures.
- Using publicly available benchmark datasets to ensure comparability with existing solutions.

However, the following aspects are considered out of scope for this study:

- Pretraining LLMs from scratch.
- Integrating the framework into real-time production systems or pipelines.
- Exploring techniques such as prompt engineering or retrieval-augmented generation (RAG) in depth.
- Performing exhaustive hyperparameter optimization for each model.

3.3 Research Questions

Given the problems outlined in Section 3.1 and the desired solution outlined in Section 3.2, this research explores the potential of LLM-based approaches to improve log analysis. The study is guided by the following key questions:

RQ1 How effective are Large Language Models (LLMs) in detecting anomalies in log sequences compared to traditional techniques?

RQ2 How does the choice of underlying LLM affect the performance of the log analysis framework?

3.4 Methodology

To address the proposed research questions, a complete pipeline was implemented. First, the raw logs from the selected dataset are split into two files: one containing only normal log sequences and another containing only abnormal log sequences. A portion of the normal sequences (5,000 sequences) is then used for training the model. Once training is complete, the model is saved and subsequently used to evaluate the remaining log sequences in order to assess its effectiveness.

It is crucial that this pipeline is tested with multiple models of varying sizes and applied to different datasets from diverse sources. In particular, it is important to use the same datasets commonly employed in other studies to enable fair and meaningful comparisons.

3.5 Framework overview

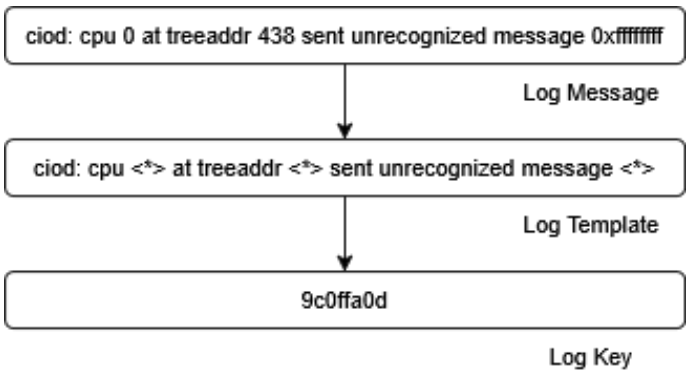

Figure 3.1: Preprocessing of log messages.

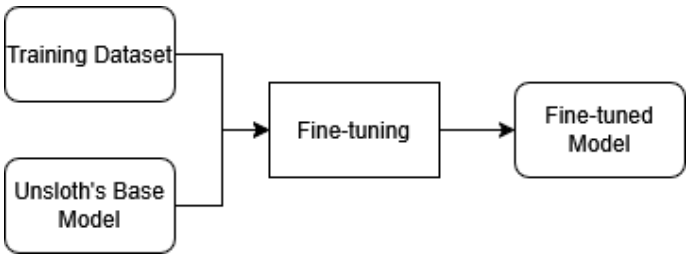

Figure 3.2: Fine-tuning of the model.

The proposed framework is designed to be *model-agnostic*, distinguishing it from existing solutions that rely on large language models (LLMs) for log anomaly detection. It is built upon

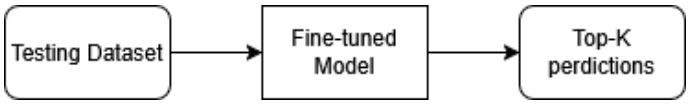

Figure 3.3: Inference using the fine-tuned model.

the Unsloth library [3], which abstracts the underlying model architecture by offering a standardized interface for loading, fine-tuning, and running various open-source LLMs such as LLaMA, Gemma, and Qwen [30, 29, 7].

Unlike frameworks tightly coupled to a specific model, such as LogGPT [10], which is built around GPT-2, this solution enables dynamic model loading. Unsloth handles the necessary backend integrations, allowing researchers to switch models by simply modifying a configuration parameter. This design fosters flexibility and future-proofing as new models become available.

The framework is composed of three main components: log preprocessing, model fine-tuning, and inference. These components are designed to be independent of the underlying model, promoting modularity. As illustrated in Figure 3.1, the preprocessing phase begins with raw log data, which is first parsed using the Drain algorithm [11] to extract structured log templates. These templates are then converted into unique log keys using a hash function. This phase is used for both the training and testing datasets, with the log sequences created from the log keys according to the rules established by the dataset providers. Next, in the fine-tuning phase (Figure 3.2), the model is trained on sequences of normal log data, allowing it to learn the typical patterns of non-anomalous behavior. Finally, during inference (Figure 3.3), the model processes sequences from the test dataset and generates Top-K predictions for each log key in the sequence. If the actual log key is not among the Top-K predictions for a position in the sequence, the entire sequence is flagged as anomalous.

Compared to tightly integrated solutions such as LogGPT and LogLLaMA [37], this framework enables greater experimentation, easier upgrades, and reduced technical debt when adapting to new models or architectures.

3.6 Preprocessing

Prior to training, a preprocessing pipeline is executed, which includes log parsing, log key generation, and log sequence construction. This process is based on the preprocessing code from LogGPT [10].

Log parsing is performed using the Drain algorithm [11], which builds a fixed-depth parse tree to group log messages into templates. Each template is then passed through a hashing function, specifically, the MD5 algorithm, to generate a unique identifier. The first 8 hexadecimal characters of the hash are used as the log key, providing a compact and consistent representation of each template. These keys are then organized in temporal order to construct sequences, which are used as input for training and for inference, and the way this is done is usually suggested by the providers of the dataset or using techniques utilized by the previous solutions for this problem.

This entire preprocessing procedure is applied to all datasets, generating corresponding log sequences from every log entry. After this step, the dataset is split into three separate files: one containing 5,000 normal log sequences used for training, where 20% will be used as the validation dataset, another with the remaining normal sequences used for testing, and a third containing all abnormal log sequences, also used for testing.

3.7 Configurations

Listing 3.1: Configuration for the framework

```
1 config = {
2     "model_name": "unsloth/gemma-3-1b-it-unsloth-bnb-4bit",
3     "max_seq_length": 128,
4     "valid_ratio": 0.2,
5     "dataset_path": "5k_hdfs_train.txt"
6 }
```

Before running the framework, it is necessary to define a configuration dictionary, as shown in the listing above. The most important parameter is `model_name` (line 2), which specifies the model to be used. Any model from the Unsloth library [3] can be selected here, allowing flexibility in choosing from a variety of popular open-source large language models.

Additional configuration options include `max_seq_length` (line 3), which defines the maximum allowed length of each log sequence, and `valid_ratio` (line 4), which determines the proportion of the training dataset to be reserved for validation. This validation subset is later used to define the value of K and to stop the training if needed. Lastly, `dataset_path` (line 5) specifies the path to the training file containing the log sequences composed of log keys.

3.8 Tokenizer

A tokenizer is a fundamental tool in any natural language processing pipeline. It enables the deconstruction of a sentence into smaller components, typically referred to as tokens. In the context of this work, the tokenizer is used to split each log sequence into its corresponding log keys.

Listing 3.2: Creation of the vocabulary for the tokenizer

```
1 all_logkeys = set()
2 with open(config["dataset_path"], "r") as f:
3     for line in f:
4         logkeys = line.strip().split()
5         all_logkeys.update(logkeys)
6
7 special_tokens = ["<bos>", "<eos>", "<pad>", "<unk>"]
8 vocab = special_tokens + list(all_logkeys)
```

```
9
10 vocab_dict = {token: i for i, token in enumerate(vocab)}
```

To build this tokenizer, the first step is to construct a custom vocabulary. Although most base language models come with a built-in tokenizer, these are typically designed for general-purpose natural language and are not suited for structured log data. Since the objective here is to tokenize only log keys, it is both sufficient and more efficient to create a specialized tokenizer using only the vocabulary present in the dataset.

To achieve this, all unique log keys are extracted from the training dataset and stored in a set (lines 1–5). Then, a set of special tokens is added to the vocabulary (line 7), including:

- <bos>: beginning-of-sequence token
- <eos>: end-of-sequence token
- <pad>: padding token
- <unk>: unknown token for unseen keys

These special tokens help manage sequence formatting and edge cases, such as handling logs of varying lengths or encountering previously unseen log keys at inference time.

Listing 3.3: Creation of the tokenizer with the vocabulary

```
1  raw_tokenizer = Tokenizer(models.WordLevel(vocab=vocab_dict, unk_token="<
       unk>"))
2
3  raw_tokenizer.pre_tokenizer = pre_tokenizers.WhitespaceSplit()
4
5  raw_tokenizer.normalizer = normalizers.Sequence([])
6
7  raw_tokenizer.post_processor = processors.TemplateProcessing(
8      single="<bos> $A <eos>",
9      special_tokens=[("<bos>", vocab_dict["<bos>"]), ("<eos>", vocab_dict["
           <eos>"])],
10 )
11
12 tokenizer = PreTrainedTokenizerFast(
13     tokenizer_object=raw_tokenizer,
14     bos_token="<bos>",
15     eos_token="<eos>",
16     pad_token="<pad>",
17     unk_token="<unk>",
18     model_max_length=config["max_seq_length"]
19 )
```

Once the vocabulary is defined, the tokenizer is created using the Hugging Face tokenizers and transformers libraries [33] (line 1). These tools provide a convenient and highly customizable way to build tokenizers.

Tokenization is performed using whitespace splitting (line 3), which is appropriate for this structured format, and no normalization steps are applied (line 5) to preserve the integrity of the log keys. A post-processing template (lines 7–10) is then defined to automatically insert the <bos> and <eos> tokens around each sequence. Finally, the tokenizer is wrapped in a PreTrainedTokenizerFast object (lines 12–19), specifying all special tokens and the maximum sequence length.

This approach ensures that the tokenizer remains lightweight, efficient, and tailored specifically to log data. After this, the tokenizer is used to tokenize all log sequences from the training dataset.

3.9 Loading the Model

Listing 3.4: Loading the underlying model with Unsloth

```
1  model, _ = FastModel.from_pretrained(
2      config["model_name"],
3      max_seq_length=config["max_seq_length"],
4      dtype=torch.bfloat16,
5  )
6
7  model.resize_token_embeddings(len(tokenizer))
8
9  model = FastModel.get_peft_model(
10     model,
11     r=32,
12     lora_alpha=64,
13     inference_mode=False,
14 )
```

Before training, the underlying model must be loaded. This is done in lines 1–6 using the `FastModel.from_pretrained` method, which takes the model name and maximum sequence length defined in the configuration. The data type used is `torch.bfloat16`, which is supported by the NVIDIA A100 GPU used for training and enables faster and more memory-efficient computation.

Since a custom tokenizer is employed, the model's input token embedding matrix must be resized to match the new vocabulary size, as shown in line 8.

Finally, to enable efficient fine-tuning, Unsloth's implementation of Low-Rank Adaptation (LoRA) [12] is applied in lines 10–15. Instead of fine-tuning all model parameters, LoRA updates only specific low-rank matrices, significantly reducing the computational load. The values for LoRA rank (r = 32) and LoRA alpha (`lora_alpha = 64`) are selected based on the Unsloth

documentation. Additionally, `inference_mode` is set to False to indicate that the model is being prepared for training rather than inference.

3.10 Fine-tuning

The model is fine-tuned using sequences of normal log keys. The goal is to train the model to learn the typical structure and patterns of normal behavior so that it can later identify deviations as anomalies.

The cross-entropy loss function, commonly used for language modeling tasks, is applied to the log sequences. It is defined as:

$$\mathscr{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i-1}\log p(k_{t+1}^i|k_{1:t}^i;\theta)$$

where θ represents the model parameters, N is the number of training sequences, T_i is the length of sequence i, and $p(k_{t+1}^i|k_{1:t}^i;\theta)$ is the probability assigned by the model to the correct next key given the previous ones. This loss function is consistent with that used in LogGPT [10].

Listing 3.5: Fine-tuning using Hugging Face's Trainer

```
early_stopping_callback = EarlyStoppingCallback(early_stopping_patience=3)

trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_tokenized_dataset,
    eval_dataset=valid_tokenized_dataset,
    callbacks=[early_stopping_callback],
    args=SFTConfig(
      output_dir="./output",
      eval_strategy="epoch",
      save_strategy="epoch",
      learning_rate=2e-4,
      num_train_epochs=100,
      per_device_train_batch_size=1028,
      per_device_eval_batch_size=1028,
      metric_for_best_model="eval_loss",
      bf16=True,
    ),
)


trainer.train()
```

The fine-tuning process is performed using the `SFTTrainer` from Hugging Face [33]. Training is carried out for a maximum of 100 epochs, following the same configuration used in the

LogGPT approach [10]. To prevent overfitting and reduce unnecessary computation, an early stopping mechanism is employed: if the validation loss does not improve for three consecutive epochs, training is stopped. The validation loss is computed at the end of each epoch using the designated loss function. The learning rate is set to 2e-4, which is the recommended starting point according to Unsloth's documentation. After training, the model from the epoch with the smallest validation error and the tokenizer are saved.

3.11 Inference

The inference phase constitutes the final step of the framework. In this stage, log sequences are evaluated by the trained model to determine whether they represent normal behavior or indicate an anomaly.

Initially, each log entry from the testing datasets undergoes the preprocessing steps described in Section 3.6. After preprocessing, the sequence of log keys is fed to the model one token at a time. The model then predicts the next log key in the sequence and compares this prediction to the actual next key.

Relying solely on the single most probable prediction would be unreliable due to the nature of log sequences, as it is common for multiple valid events to follow any given one. For example, after a user logs into a system, the next event could be opening a dashboard, retrieving a profile, or starting a background sync. All of these are valid continuations, and limiting the model to a single prediction would lead to frequent false positives.

To address this, the framework considers the Top-K most probable next log keys, as predicted by the model. For each step in the sequence, the true next log key is checked against this Top-K set. If it is present, the step is considered normal, otherwise, it is flagged as anomalous.

A sequence is classified as anomalous if at least one of its log keys is not included in the corresponding Top-K predictions. On the other hand, if all log keys in the sequence are consistently found within the Top-K predictions, the sequence is considered normal.

3.11.1 Selecting K

To determine an appropriate value for K in the Top-K inference phase, the validation dataset, derived from the training dataset that only contains normal sequence of logs, is used. Specifically, the smallest K is chosen such that, for every position in every sequence within the validation set, the true next log key appears in the model's Top-K predictions. This translates to the first K with which the model considers all the sequences from the validation dataset as normal sequences, as it should. This ensures that the chosen value of K captures normal behavior patterns as reliably as possible before testing the model on unseen data.

Listing 3.6: Function to select the K

```
1 def calculate_multiple_topk_miss_rates(sequence, topk_candidates):
2     model.eval()
```

```
    device = model.device

    miss_rates_by_k = {k: [] for k in topk_candidates}

    with torch.no_grad():
        with tqdm(total=len(sequence), desc=f"Evaluation", unit="sequence"
            ) as pbar:
            for seq_idx, token_list in enumerate(sequence):
                token_tensor = torch.tensor(token_list, dtype=torch.long,
                    device=device).unsqueeze(0)
                input_ids = token_tensor[:, :-1]
                labels = token_tensor[:, 1:]

                if input_ids.size(1) == 0:
                    for k in topk_candidates:
                        miss_rates_by_k[k].append(1.0)
                    pbar.update(1)
                    continue

                with torch.amp.autocast(device_type='cuda', dtype=torch.
                    float16):
                    outputs = model(input_ids=input_ids)

                logits = outputs.logits.to(torch.float32)

                total_tokens_to_predict = labels.size(1)

                if total_tokens_to_predict <= 2:
                    for k in topk_candidates:
                        miss_rates_by_k[k].append(0.0)
                    pbar.update(1)
                    continue

                max_k = max(topk_candidates)
                _, topk_predictions = torch.topk(logits, k=max_k, dim=-1)

                for k in topk_candidates:
                    correct = 0
                    for i in range(total_tokens_to_predict):
                        true_token_id = labels[0, i].item()
                        predicted_k_token_ids = topk_predictions[0, i, :k
                            ].tolist()
                        if true_token_id in predicted_k_token_ids:
                            correct += 1
                    miss_rate = 1.0 - (correct / total_tokens_to_predict)
```

```
                    miss_rates_by_k[k].append(miss_rate)
                pbar.update(1)
    return {k: np.array(miss_rates) for k, miss_rates in miss_rates_by_k.
        items()}
```

To select the appropriate value for K, the function presented in the Listing 3.6 is used. This function calculates the number of misses, which correspond to instances where the model fails to include the true next log key within its top-K predictions.

To improve efficiency, the function avoids querying the model separately for each candidate and each position in the sequence. Instead, it retrieves the top-M predictions, where M is the largest value among the provided candidates, for every log key in the sequence. The top predictions for smaller candidate values are then obtained by slicing the top-M results, avoiding redundant computation.

This method significantly increases efficiency, as the naive approach would require a separate forward pass through the model for each candidate at every position in every sequence. This results in a computational complexity of approximately $O(N \cdot C \cdot L)$ model calls, where N is the number of sequences, C is the number of candidates, and L is the number of tokens per sequence. In contrast, the implemented function improves upon the naive approach by significantly reducing redundant computations. Instead of performing a model inference for each candidate value of K and for each position, the model's forward pass is executed only once per sequence. This single pass generates the full set of logits, the raw output scores for every possible next token at each position. From these logits, it is possible to identify the top-M predictions for every position, where M is the largest candidate value. The predictions for all smaller candidate values are then obtained by simply slicing this pre-computed top-M result. This method decouples the expensive model inference step from the number of K candidates, leading to a substantial improvement in efficiency, particularly for long sequences and a large set of candidate values, resulting in a computational complexity of approximately $O(N)$ model calls.

The function returns a dictionary mapping each candidate to its corresponding miss rate over the validation dataset. To determine the most appropriate K, the smallest candidate with a miss rate of zero is selected. This ensures that all log keys across all sequences in the validation set appear within the model's top-K predictions, meaning the model successfully recognizes all normal patterns from the validation dataset.

3.12 Testing

For the testing phase, the two remaining datasets are taken into account. One containing only normal log sequences and another containing abnormal log sequences. For this phase, a similar function to the one described in Section 3.11.1 is used, with key variations to accommodate the larger scale of the testing datasets and the fixed nature of the K value already determined in the selection phase.

Listing 3.7: Function to test the trained model

```
def calculate_topk_miss_rate_optimized(sequences, model, tokenizer, top_k,
    batch_size):
    device = model.device
    model.eval()
    all_miss_rates = []

    pad_token_id = tokenizer.pad_token_id or tokenizer.eos_token_id

    with torch.no_grad():
        for i in tqdm(range(0, len(sequences), batch_size), desc=f"Top-{
            top_k} Miss Rate"):
            batch_seqs = sequences[i:i + batch_size]
            batch_tensors = [torch.tensor(seq, dtype=torch.long) for seq
                in batch_seqs]

            valid_indices = [idx for idx, t in enumerate(batch_tensors) if
                len(t) > 2]
            if not valid_indices:
                all_miss_rates.extend([0.0] * len(batch_tensors))
                continue

            batch_tensors = [batch_tensors[idx] for idx in valid_indices]

            input_ids = [t[:-1] for t in batch_tensors]
            labels = [t[1:] for t in batch_tensors]

            input_ids = pad_sequence(input_ids, batch_first=True,
                padding_value=pad_token_id).to(device)
            labels = pad_sequence(labels, batch_first=True, padding_value=
                pad_token_id).to(device)
            attention_mask = (input_ids != pad_token_id).long()

            with torch.amp.autocast(device_type=device.type, dtype=torch.
                float16 if device.type == 'cuda' else torch.float32):
                logits = model(input_ids=input_ids, attention_mask=
                    attention_mask).logits

            logits = logits.to(torch.float32)

            if tokenizer.unk_token_id is not None:
                logits[:, :, tokenizer.unk_token_id] = float('-inf')

            topk_preds = torch.topk(logits, k=top_k, dim=-1).indices  # [B
                , L, K]
```

```
36
37              labels_exp = labels.unsqueeze(-1)
38              correct = (topk_preds == labels_exp).any(dim=-1)
39              valid = labels != -100
40
41              hits = (correct & valid).sum(dim=1)
42              total = valid.sum(dim=1)
43              miss_rate = 1.0 - (hits.float() / total.float())
44
45
46              result = [0.0] * len(batch_seqs)
47              for j, idx in enumerate(valid_indices):
48                  result[idx] = miss_rate[j].item()
49
50              all_miss_rates.extend(result)
51
52      return np.array(all_miss_rates)
```

The core functionality of this function remains identical to the K selection function, as it calculates the number of misses, the number of times the model fails to include the next true log key in the top-K predictions. However, several adaptations were made to address the different requirements of the testing phase.

First, the K value is fixed using the number determined by Section 3.11.1, so there is no need to evaluate multiple candidate values, and no need to slice the results according to different K values. The list of predictions resulting from the query to the model is already of the required length K, eliminating the efficiency optimization of computing top-M predictions and slicing for smaller values.

The most significant difference is the implementation of batch processing. To enable this process, and because sequences may have different lengths, the function pads the sequences accordingly. With the utilization of the `pad_sequence` function, all sequences are padded to the same length using the padding token from the tokenizer, creating uniform tensor dimensions required for batch operations. An attention mask is also generated to ensure the padding tokens are ignored during inference. This batched approach reduces the computational complexity from O(N) individual model calls to O(N/B) batch calls, where B is the batch size, providing substantial performance improvements for the larger testing datasets.

For calculating the miss rates, the function uses vectorized tensor operations to compute hits and misses across all positions in all sequences in the batch simultaneously, then combines individual token misses into miss rates for entire sequences. The return format is also different from the selecting K function, where this time a simple numpy array is returned containing the miss rate for each sequence.

The resulting miss rates serve as the foundation for anomaly detection. For the first testing dataset containing only normal sequences, all non-zero values are considered false positives (nor-

mal sequences incorrectly flagged as anomalous), and all zero values are considered true negatives (normal sequences correctly identified as normal). For the second testing dataset containing only abnormal log sequences, all non-zero values are considered true positives (abnormal sequences correctly identified as anomalous), while all zero values are considered false negatives (abnormal sequences incorrectly classified as normal).

Chapter 4

Experimental Work

This chapter introduces the experiments designed to answer the research questions. It begins by outlining the experimental setup, followed by the datasets and underlying models, and then a description of the experiments themselves. Finally, it presents the analysis of the results and explains how they address the research questions.

4.1 Experimental Setup

The experiments were conducted in two different setups: one for training and another for inference. This separation allows for rapid experimentation with different models, as the training phase is significantly more time-consuming than inference due to the need to iterate over the dataset multiple times.

The training setup uses Google Colab, specifically the A100 environment. This GPU has 40 GB of VRAM, and the system's RAM may vary depending on Google's resource allocation at the time.

The inference setup was run on a laptop equipped with an Intel Core i7-10750H CPU @ 2.60GHz, an NVIDIA GeForce RTX 2060 GPU, and 16 GB of RAM. All experiments on this laptop were performed while connected to a power source.

4.2 Datasets

To evaluate our framework, three widely adopted datasets in the log anomaly detection domain [13] were used. Leveraging these well-established benchmarks enables fair and direct comparisons with other state-of-the-art approaches. The selected datasets are HDFS, BGL, and Thunderbird, described below:

- **HDFS (Hadoop Distributed File System)**: This dataset comprises logs from the Hadoop Distributed File System, a system designed for managing large-scale data storage and processing. The logs were originally collected in 2008 and labeled by domain experts [35], on

a Hadoop cluster comprised of more than 200 nodes. Log messages are grouped into log sequences based on session IDs.

- **BGL (BlueGene/L)**: Collected from the BlueGene/L supercomputer at Lawrence Livermore National Laboratory, this dataset was introduced in [23], where the first logs were collected of 32768 dual-processor nodes and then was upgraded to 65536 nodes during the log collection between 2005 and 2006. Log sequences are constructed using a sliding time window of 60 seconds.

- **Thunderbird**: Also presented in the same work as BGL [23], this dataset originates from systems at Sandia National Laboratories. For consistency with prior research, a subset containing 20 million log messages was selected. As with BGL, log sequences are created using a 60-second sliding time window.

Table 4.1 summarizes the dimensions of each dataset. The dataset splits align with those used in LogGPT [10], allowing for meaningful comparisons.

Table 4.1: Dataset Statistics

Dataset	**Training Data**	**Normal Test Data**	**Anomalous Test Data**
HDFS	5,000	553,223	16,839
BGL	5,000	28,631	3,296
Thunderbird	5,000	67,039	40,920

4.3 Models

To demonstrate the flexibility of the model-agnostic framework, experiments were conducted using three different lightweight LLMs available through the Unsloth library [3]:

- **Gemma 3 (1 billion parameters)** [29]
- **LLaMA 3.2 (1 billion parameters)** [7]
- **Qwen 2.5 (0.5 billion parameters)** [30]

Gemma 3 is a set of efficient, open-source models developed by Google DeepMind. The smallest variant from this family was selected to ensure compatibility with lower-end hardware.

LLaMA 3.2, developed by Meta, is a compact model series derived from the original 8-billion-parameter LLaMA 3.1 models. Through pruning and distillation, it has been optimized for resource-constrained environments, including mobile deployment.

Qwen 2.5, released by Alibaba, is a collection of open-source models designed with efficiency in mind. The smallest version, featuring 0.5 billion parameters, is well-suited for lightweight applications such as edge or mobile inference.

The focus of these experiments was on models capable of running on most modern consumer hardware. Although powerful GPUs can accelerate training, the resulting models are lightweight enough for inference on standard systems.

4.4 Baselines

The proposed framework is compared against traditional machine learning methods, deep learning methods (such as LSTMs), and transformer-based methods. The latter category includes two of the most prominent existing approaches that utilize Large Language Models (LLMs) for log anomaly detection.

- **PCA** [34]: Detects anomalies by applying Principal Component Analysis to structured log feature vectors and identifying deviations from normal patterns.
- **iForest** [16]: Uses randomly constructed isolation trees to score anomalies based on how easily data points can be isolated.
- **SVM** [32]: Applies kernel-based one-class classification to capture structural log patterns and detect anomalies.
- **DeepLog** [5]: DeepLog uses LSTM networks to model normal system behavior by predicting the next log key in a sequence and estimating parameter values and delays between log messages. It combines structural and timing information to detect anomalies in execution paths and performance metrics.
- **LogAnomaly** [19]: LogAnomaly enhances LSTM-based anomaly detection by introducing Template2Vec, which embeds log templates with semantic information using synonym and antonym relationships. It captures both sequential and quantitative patterns in log sequences and flags anomalies based on top-k predictions. The framework also handles unseen log types by approximating them with the most similar known templates.
- **LogBERT** [8]: LogBERT applies Transformer-based modeling to log sequences using masked log key prediction and hypersphere volume minimization to learn robust representations of normal behavior. Anomalies are detected when a log sequence deviates from learned patterns, with predictions evaluated based on how many tokens fall outside the top-g predictions.
- **LogGPT** [10]: LogGPT is based on GPT-2 and is trained using the same loss function adopted in the present framework, with the objective of generating normal log sequences. For Top-K prediction, the value of K is set to half the number of unique log keys in the training set.

- **LogLLaMA** [37]: LogLLaMA employs LLaMA 2 as its base model, although the specific version is not disclosed in the original publication. Its training methodology closely mirrors that of LogGPT, with a key difference being the amount of training data used, where instead of a fixed 5,000 log lines, LogLLaMA utilizes 10% of the full dataset, resulting in significantly larger training data. It is important to note that, at the time of writing, the code for this solution has not been made publicly available. As a result, it is not possible to verify the implementation details or independently validate the results reported in the paper. Nonetheless, the reported results will still be considered and compared with those obtained using the proposed framework.

4.5 Evaluation Metrics

The selected datasets exhibit a high degree of class imbalance, with normal log sequences vastly outnumbering anomalous ones. This imbalance mirrors real-world scenarios, where system anomalies are relatively rare. As a result, and in line with prior work, model performance is evaluated using Precision, Recall, and, most critically, the F1 Score.

The evaluation metrics are defined as follows:

$$\text{Precision} = \frac{TP}{TP+FP} \tag{4.1}$$

$$\text{Recall} = \frac{TP}{TP+FN} \tag{4.2}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

4.6 Results and Analysis

Table 4.2 as well as the charts in Figure 4.1, Figure 4.2, and Figure 4.3, summarize the performance of a variety of log anomaly detection methods across three benchmark datasets: HDFS, BGL, and Thunderbird. These methods include traditional statistical and machine learning approaches (PCA, iForest, SVM), deep learning-based solutions (DeepLog, LogAnomaly, LogBERT), as well as recent large language model (LLM) based approaches such as LogGPT, LogLLaMA, and finally the framework presented in this thesis with the following underlying models: Qwen 2.5, Gemma 3, and Llama 3.2.

The traditional machine learning approaches generally exhibit limited effectiveness across the datasets. PCA demonstrates particularly poor performance in both the HDFS and BGL datasets, with very low recall values, indicating a failure to identify the majority of anomalies. In contrast, it performs surprisingly well on the Thunderbird dataset, achieving a high F1 score, possibly due to the simplicity or regularity of the log patterns in that dataset.

Table 4.2: Results in the HDFS, Thunderbird, and BGL Datasets.

	HDFS			**BGL**			**Thunderbird**		
	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score
PCA	0.166	0.059	0.087	0.117	0.035	0.054	0.953	0.980	0.966
iForest	0.043	0.422	0.078	0.491	0.037	0.063	0.338	0.015	0.028
SVM	0.058	0.910	0.108	0.073	0.345	0.121	0.550	0.998	0.709
DeepLog	0.793	0.863	0.824	0.792	0.946	0.861	0.864	0.997	0.926
LogAnomaly	0.907	0.369	0.524	0.884	0.850	0.867	0.873	0.996	0.931
LogBERT	0.754	0.749	0.745	0.917	0.892	0.905	0.962	0.965	0.963
LogGPT	0.884	0.921	0.901	0.940	0.977	0.958	**0.973**	**1.000**	**0.986**
LogLLaMA	**0.939**	0.852	0.894	0.927	**0.993**	0.959	0.957	0.992	0.974
Qwen 2.5	0.881	0.734	0.801	0.937	0.985	0.96	0.971	0.996	0.983
Gemma 3	0.905	0.919	0.912	**0.970**	0.984	**0.977**	0.971	0.997	0.984
Llama 3.2	0.883	**0.986**	**0.932**	0.933	0.987	0.959	0.971	0.996	0.983

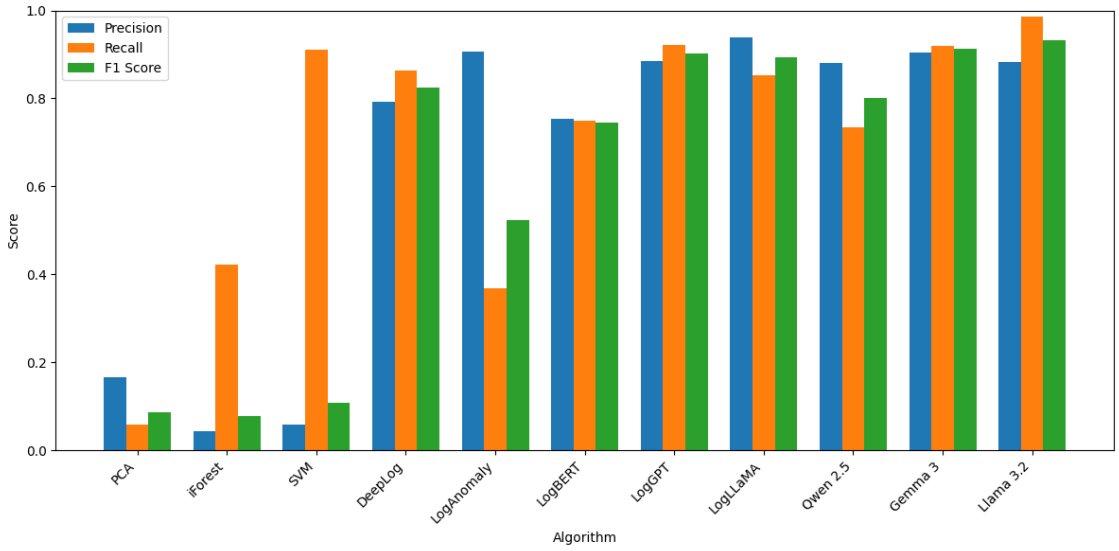

Figure 4.1: Performance of Algorithms on the HDFS Dataset: Precision, Recall, and F1 Score

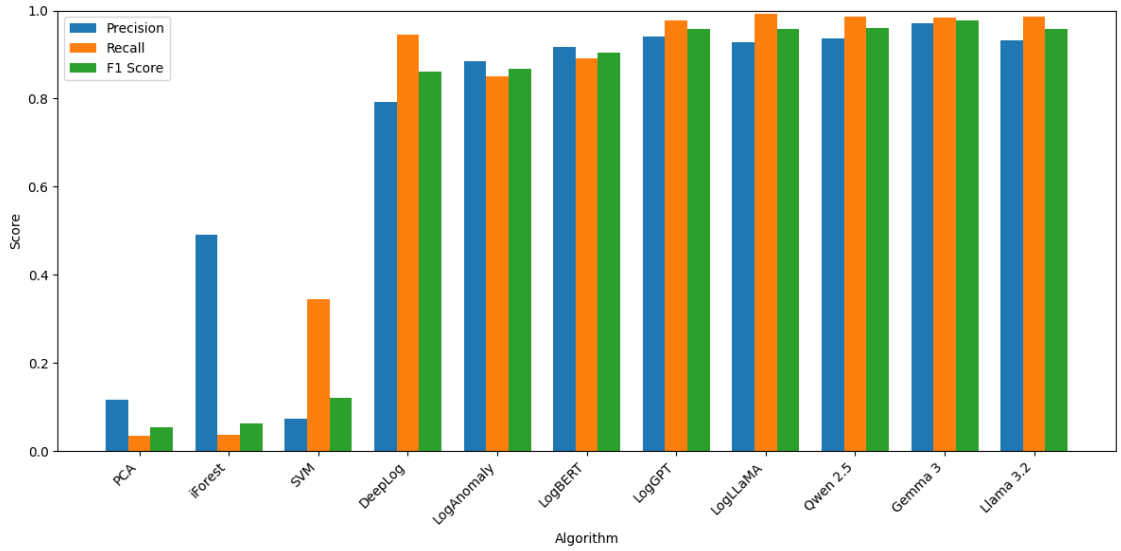

Figure 4.2: Performance of Algorithms on the BGL Dataset: Precision, Recall, and F1 Score

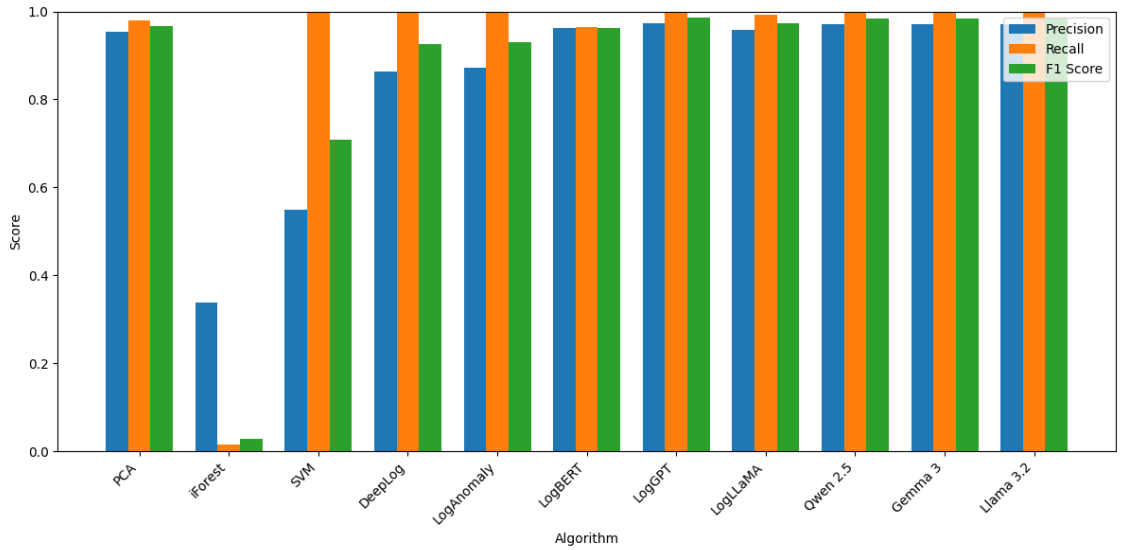

Figure 4.3: Performance of Algorithms on the Thunderbird Dataset: Precision, Recall, and F1 Score

Isolation Forest (iForest) achieves bad results all around and is, in general, the worst-performing method from the evaluated ones. SVM performs inconsistently across datasets, while exhibiting high recall on HDFS and Thunderbird but producing low precision, suggesting a tendency toward over-detection of anomalies. These results indicate that this method may struggle to balance sensitivity and specificity in complex or noisy log environments.

Among the earlier deep learning approaches, DeepLog performs consistently across all datasets, achieving high F1 scores that surpass those of traditional methods, with the exception of the Thunderbird dataset when compared with the PCA method. While LogAnomaly, the other method that leverages LSTMs, shows mixed results, as it yields strong precision on HDFS, but poor recall, resulting in a modest F1 score. These two methods are very similar in performance, with the exception of the HDFS dataset, where DeepLog takes big advantage. However, when compared with the other deep learning methods, they fall behind in every dataset.

LogBERT, a transformer-based model, particularly excels on Thunderbird while achieving near state-of-the-art results in the BGL datasets. However, it has a very poor performance in the HDFS dataset.

The more recent LLM-based methods, LogGPT, LogLLaMA, and the framework with Qwen 2.5, Gemma 3, and Llama 3.2, demonstrate significantly stronger performance, especially in terms of F1 score. LogGPT and LogLLaMA already establish high baselines across datasets. However, the inclusion of newer models in a model-agnostic framework leads to further improvements.

Llama 3.2 obtains the highest F1 score on the HDFS dataset and remains highly competitive on BGL and Thunderbird. Gemma 3 achieves the best overall F1 score on the BGL dataset and matches Llama 3.2 on Thunderbird. Qwen 2.5, while competitive on BGL and Thunderbird, lags behind on HDFS, with notably low recall. This drop in performance is likely attributable to its reduced parameter size, which may limit its ability to capture complex dependencies between log keys.

Despite LogGPT slightly outperforming newer models on Thunderbird, the performance gap is marginal, and nearly all LLM-based methods reach excellent scores on this dataset. This suggests that Thunderbird is a less complex benchmark, where both older and newer methods are able to generalize well.

These findings underscore the substantial performance gap between LLM-based methods and traditional or early deep learning approaches in log anomaly detection. They also highlight the importance of selecting an appropriate model for the characteristics of each dataset, supporting the need for flexible frameworks that enable users to switch between models depending on their resource constraints and desired performance trade-offs.

These results illustrate the practical advantages of adopting a model-agnostic framework. For instance, the ability to test with a smaller model, such as Qwen 2.5, can be highly beneficial in resource-constrained environments. While its performance on the HDFS dataset is limited, it achieves near state-of-the-art results on the Thunderbird and BGL datasets, making it a viable and cost-effective choice in scenarios where computational efficiency is prioritized.

Furthermore, the framework enables seamless switching between high-performing models like Gemma 3 and Llama 3.2, depending on the dataset characteristics. This flexibility allows users to consistently select the most suitable model for their specific use case, such as Llama 3.2 for HDFS and Gemma 3 for BGL, without requiring changes to the surrounding infrastructure. This adaptability is essential for real-world deployments, where the nature of log data and anomaly patterns can vary significantly between systems.

4.7 Research Questions Remarks

The experiments conducted aimed to answer the research questions introduced in Section 3.3. Below are the answers that can be drawn from the results of this experimental phase:

RQ1 How effective are Large Language Models (LLMs) in detecting anomalies in log sequences compared to traditional techniques?

Overall, LLMs appear to be the most effective method for detecting anomalies, as they consistently achieve the highest F1 scores across all datasets. Both the state-of-the-art solutions and the proposed framework outperform traditional methods in every dataset. This performance gap is especially evident in the HDFS and BGL datasets. The Thunderbird dataset yields high F1 scores across almost all methods, likely because its log patterns are simpler and more easily identifiable.

RQ2 How does the choice of underlying LLM affect the performance of the log analysis framework?

The results show that the choice of LLM has a significant impact on performance. No single model achieves the highest F1 score across all datasets. Llama performs best on HDFS, Gemma on BGL, and the GPT model used in the LogGPT framework performs best on Thunderbird. While Gemma generally delivers strong results across datasets, using Llama on HDFS, for example, offers a noticeable performance advantage. This highlights the importance of being able to switch the underlying model.

Chapter 5

Conclusions

The previous chapter presented the results and analysis of the proposed framework's utilization. This chapter focuses on the conclusions of the work developed.

5.1 Conclusions

The importance of observability in today's complex and distributed systems has grown significantly. It plays a critical role in ensuring system reliability, maintainability, and performance. Observability is commonly structured around three pillars: traces, metrics, and logs. This thesis focuses specifically on the log pillar, more precisely, on how log data, particularly log sequences, can be analyzed automatically and effectively in order to extract meaningful insights and support timely decision-making.

Log analysis has long been the subject of academic research, with a wide range of techniques proposed to tackle its inherent challenges. Early approaches were based on traditional machine learning models, including methods such as Principal Component Analysis (PCA), Isolation Forests, and Support Vector Machines (SVM) [34, 16, 32]. However, these techniques have since been largely surpassed by the emergence of neural network-based solutions, which offer greater accuracy for processing sequential data.

Notably, Long Short-Term Memory (LSTM) networks were among the first neural models to be applied to log analysis, as seen in works such as [5, 19]. These models introduced key ideas, such as training exclusively on normal log sequences to identify anomalies by modeling expected behavior, which continue to influence more recent developments.

Building upon this foundation, the field has recently seen a shift toward the use of Transformer-based architectures [31], for example in LogBERT [8], and particularly with the rise of Large Language Models (LLMs), which have gained popularity following the public release of tools like ChatGPT. LLM-based solutions such as LogGPT and LogLLaMA [10, 37] leverage the powerful contextual understanding of these models to push the boundaries of automated log analysis.

In response to these advancements, this thesis introduces a model-agnostic log analysis framework designed to support experimentation with different LLMs. The framework decouples the

underlying model from the rest of the system, allowing users to easily swap in different LLMs depending on the specific characteristics of their log data. This flexibility is particularly valuable given the rapid pace at which new LLMs are being released and improved, enabling organizations to adapt their analysis pipelines without needing to overhaul their entire infrastructure.

The framework was thoroughly evaluated using three widely adopted benchmark datasets in the field of log anomaly detection: HDFS, BGL, and Thunderbird. The results demonstrate that the proposed solution not only achieves competitive performance but also outperforms state-of-the-art methods in several cases.

Specifically, on the HDFS dataset, the framework achieved F1 scores of 0.932 with LLaMA 3.2 and 0.912 with Gemma 3 , both surpassing the previously best result from the current state-of-the-art in this field LogGPT (F1: 0.901).

On the BGL dataset, the framework attained an F1 score of 0.977 with Gemma 3, exceeding all existing solutions. In addition, Qwen 2.5 achieved a strong performance with an F1 score of 0.960, also outperforming the state-of-the-art for this dataset LogLLaMA (F1: 0.959).

In the Thunderbird dataset, while LogGPT remains the top performer with an F1 score of 0.986, the proposed framework came remarkably close, achieving F1 scores of 0.984 with Gemma 3, and 0.983 with both Qwen 2.5 and LLaMA 3.2, demonstrating only a very small marginal difference.

These findings underscore the effectiveness of the proposed framework and its adaptability to different LLMs, confirming that no single model is optimal across all datasets. Instead, the flexibility to select the most suitable model for each scenario proves to be a key strength, especially given the continuous advancements in LLM research. These findings reinforce the central contribution of this work: that flexibility and modularity in log analysis pipelines are essential for keeping pace with the evolving landscape of machine learning technologies. In every dataset tested, the most effective model varied, highlighting that no single LLM is universally optimal, and further justifying the need for a model-agnostic architecture. As LLM research continues to advance, the ability to easily adopt and evaluate new models will become increasingly critical for organizations seeking to maintain robust and intelligent observability solutions.

5.2 Future Work

One promising direction for future research involves exploring how the framework can adapt to evolving log patterns over time. In real-world software systems, log structures often change as new features are added or existing components are modified. Even minor updates or bug fixes can introduce previously unseen log formats, potentially affecting the model's ability to distinguish between normal and anomalous behavior. Investigating mechanisms that allow the framework to adapt to these changes dynamically would enhance its robustness and long-term usability.

Machine learning techniques such as transfer learning could be particularly beneficial in this context. For instance, a model initially trained on a large dataset could be periodically fine-tuned on newer log data, using the previously trained model as a starting point. This approach would

enable the system to retain previously learned knowledge while incorporating new log patterns, improving performance without the need to retrain from scratch.

Another avenue for enhancement involves incorporating reinforcement learning, inspired by methods used in LogGPT [10]. By integrating a feedback loop where the model receives rewards based on prediction accuracy or operational feedback, it may be possible to further refine the framework's performance over time.

Together, these techniques could significantly strengthen the adaptability and effectiveness of the framework in real-world production environments, where log data is dynamic and ever-changing.

References

[1] Saumen Biswas. Comprehensive observability and monitoring strategies using datadog. *International Journal of Management, IT and Engineering*, 15(4):15–21, 2025.

[2] Lee Boonstra. Prompt engineering, 2024.

[3] Michael Han Daniel Han and Unsloth team. Unsloth, 2023.

[4] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864, 2016.

[5] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1285–1298, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Ricardo Dunia and Joe Qin. Multi-dimensional fault diagnosis using a subspace approach. 04 1997.

[7] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[8] Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[9] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. CIKM '16, page 1573–1582, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Xiao Han, Shuhan Yuan, and Mohamed Trabelsi. Loggpt: Log anomaly detection via gpt, 2023.

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[12] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

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

[14] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[15] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 102–111, 2016.

[16] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.

[17] Ankur Mahida. Enhancing observability in distributed systems-a comprehensive review. *Journal Of Mathematical & Computer Applications. Src/Jmca-166. Doi: Doi. Org/10.47363/Jmca/2023 (2)*, 135:2–4, 2023.

[18] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2012.

[19] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

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

[21] Kim Anh Nguyen, Sabine Schulte im Walde, and Ngoc Thang Vu. Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction. *arXiv preprint arXiv:1605.07766*, 2016.

[22] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[23] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 575–584. IEEE, 2007.

[24] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. Fine-tuning or retrieval? comparing knowledge injection in llms, 2024.

[25] Robert Poenaru and Dragos Ciobanu-Zabet. Elk stack – improving the computing clusters at dfcti through log analysis. In *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–8, 2020.

[26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[27] Sreemathy S P, Vijay Kumar C, and Sugantha Priyadharshini P. Application monitoring and telemetry analytics. In *2024 7th International Conference on Circuit Power and Computing Technologies (ICCPCT)*, volume 1, pages 1559–1565, 2024.

[28] Peter HA Sneath and Robert R Sokal. Unweighted pair group method with arithmetic mean. *Numerical Taxonomy*, pages 230–234, 1973.

[29] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[30] Qwen Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[32] Y. Wang, J. Wong, and A. Miner. Anomaly intrusion detection using one class svm. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pages 358–364, 2004.

[33] 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.

[34] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 117–132, New York, NY, USA, 2009. Association for Computing Machinery.

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

[36] KeHan Xue, Qiang Han, Sheng Han, ZhiChao Shi, and YiXin Qiao. A review of software testing process log parsing and mining. In *2024 IEEE International Conference on Software Services Engineering (SSE)*, pages 334–343, 2024.

[37] Zhuoyi Yang and Ian G. Harris. Logllama: Transformer-based log anomaly detection with llama, 2025.

[38] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130, 2019.