

VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING



Anomaly Detection in HDFS Logs Using Machine Learning Integrated With LLM-Based Mitigation

By
Nguyen Hoang Quan - ITITIU21291

The thesis submitted to School of Computer Science and Engineering in partial
fulfillment of the requirements of the degree of Bachelor of
Engineering of Information Technology

Ho Chi Minh, Viet Nam

2025

Anomaly Detection In HDFS Logs Using Machine Learning Integrated With LLM-Based Mitigation

APPROVED BY:

THESIS COMMITTEE

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to the School of Computer Science and Engineering for providing a supportive academic environment and the essential resources that made this thesis possible.

I am deeply thankful to my advisor, Dr. Le Hai Duong, for his invaluable guidance, encouragement, and constructive feedback throughout the development of this thesis. His expertise and commitment has greatly enriched both my technical understanding and research direction.

I would also like to extend my appreciation to all lecturers and staff members who have contributed to my academic journey with their instruction, support, and mentorship. Special thanks go to my family and friends for their constant support, motivation, and understanding during the preparation of this work. Lastly, I would like to acknowledge the developers and the researchers behind the resources for providing the foundation upon which this project was built.

This thesis would not have been possible without the guidance, resources, and support from all of the above.

Contents

ABSTRACT	1
Chapter 1: INTRODUCTION	1
1.1 Background of the study	1
1.2 Problem Statement	2
1.3 Objectives of the Study	2
1.4 Limitations of the Study	3
Chapter 2: LITERATURE REVIEW	4
2.1 Introduction to system log	4
2.2 Traditional log analysis method	5
2.2.1 Manual Log Inspection	5
2.2.2 Rule-Based Systems (Regex and Pattern Matching)	6
2.3 Log parsing	6
2.4 Machine learning and deep learning for log anomaly detection	7
2.4.1 Supervised learning	7
2.4.2 Unsupervised learning	9
Chapter 3: METHODOLOGIES	11
3.1 Overview	11
3.1.1 Research Approach	11
3.1.2 System Pipeline Overview	12
3.2 System Architecture and Design	13
3.2.1 Architectural Overview and Design Principles	13

3.2.2	High-Level System Architecture	14
3.3	Dataset preparation	17
3.4	Log Parsing Methodology	18
3.5	Feature Extraction / Representation	19
3.6	Anomaly Detection Models	20
Chapter 4: IMPLEMENTATIONS		21
4.1	Implementation Overview	21
4.2	Implementation timeline	21
4.3	Backend Implementation	24
4.3.1	FastAPI Application Architecture	24
4.3.2	Database Layer Implementation	26
4.3.3	API Endpoints Implementation	29
4.4	Log Processing Pipeline	32
4.4.1	File Upload and Storage	32
4.4.2	Asynchronous Processing with Celery	35
4.4.3	Drain Parser Integration	38
4.4.4	Feature Extraction	39
4.4.5	Machine Learning Implementation for Anomaly Detection	40
4.5	LLM Integration	41
4.5.1	API client setup and Context preparation	42
4.5.2	Prompt Engineering	43
4.6	Frontend Implementation	44
4.6.1	React Application Structure	44
4.6.2	Key Components Implementation	45
Chapter 5: RESULT ANALYSIS		48
5.1	Machine learning models evaluations	48
5.2	LLM responses	50
5.3	User Interface Effectiveness	51

5.4	Limitations	55
Chapter 6:	CONCLUSION AND RECOMMENDATION	56
6.1	Key findings	56
6.2	Recommendations for future work	56
6.3	Conclusion of the research	57

List of Tables

4.1	Router Responsibilities	25
4.2	API endpoints	29

List of Figures

2.1	An example of raw logs taken from HDFS dataset	4
2.2	Example of Log Parsing	7
2.3	SVM classification	8
2.4	Isolation Forest	10
3.1	System pipeline	13
3.2	System architecture	15
3.3	Structure of Parse Tree in Drain (depth = 3)	18
4.1	main.py file	25
4.2	Example of a router configuration pattern	26
4.3	System database schema	27
4.4	System database schema	28
4.5	Logs upload endpoints	31
4.6	Token endpoints	32
4.7	minio_client.py file	33
4.8	File upload method	34
4.9	Objects stored in MinIO	34
4.10	Celery application configuration	35
4.11	Log process task implementation	37
4.12	Drain algorithm wrapper implementation	39
4.13	Feature extraction implementation	40
4.14	Standard training workflow	41
4.15	LLM configurations	42

4.16	LLM context preparation	43
4.17	React frontend structure	45
4.18	Sequence diagram for authentication flow	46
4.19	Sequence diagram for upload log	47
5.1	Training accuracy	49
5.2	Testing accuracy	49
5.3	Example of a LLM response	50
5.4	Use Case Diagram - AI Log Analysis Web Application	51
5.5	Login page	52
5.6	User dashboard page	52
5.7	Log uploading page	53
5.8	LLM responses page	53
5.9	User management admin page	54
5.10	File management admin page	54

ABSTRACT

Anomaly detection is essential for managing today’s large-scale distributed systems, where system logs are a key resource for identifying unusual behavior. Traditionally, system operators relied on manual inspection methods such as keyword searches and rule-based matching. However, due to the massive volume and complexity of modern system logs, manual approaches are no longer practical. To tackle this, many automated log-based anomaly detection methods have been proposed. Still, developers often struggle to choose a suitable method, as there hasn’t been a clear comparison of these approaches.

In this research, I will propose a solution to addresses the fundamental challenge of automated log analysis. Specially, the research tackles three interconnected problems: (1) automated parsing of diverse log formats into structured templates, (2) anomaly detection in high-volume log streams, and (3) generation of contextual, actionable recommendations for identified issues.

Previous research has established some foundational approaches including the algorithms for log parsing and machine learning techniques for anomaly detection. However, existing solutions typically focus on individual components rather than providing end-to-end integration. Most academic implementations lack production-ready deployment architectures, user-friendly interfaces, and the integration of modern Large Language Models (LLMs) for intelligent recommendations—creating a significant gap between theoretical algorithms and practical deployment. Therefore, this research is important since it close the gap between academic log analysis algorithms and production-ready systems.

Furthermore, the research establishes an application that integrating emerging LLM capabilities into traditional system administration workflows, suggesting broader implications for AI-assisted practices. The findings indicate that intelligent automation of log analysis is not only technically feasible but can significantly enhance organizational capabilities in system reliability, security monitoring, and operational efficiency.

CHAPTER 1: INTRODUCTION

1.1 Background of the study

In modern software systems, log files serve as critical sources of information for system monitoring, debugging, troubleshooting, and security analysis. As applications or systems scale and increase its complexity, the volume of generated log data gets bigger exponentially. Therefore, traditional manual approaches for log analysis like manually examine through log files using basic tools such as grep or text editors, has become less efficient and more defective [2]. As the result, it is becoming more difficult to detect anomalies within large scale system.

Over the year, a lot of automated log-based methods have been introduced to help detecting system anomalies. These approaches usually require raw log preprocessing techniques, feature extraction and machine-learning-based algorithms for processing vast amounts of unstructured log data efficiently, identifying potential issues before they escalate into critical failures. Recent advancements in natural language processing and large language models (LLMs) have also increased the potential for intelligent log analysis systems. However, despite these development, traditional machine-learning techniques and LLM-based approaches have yet to be efficiently integrated into a single, cohesive log analysis framework.

This study focuses on leveraging existing machine learning approaches where log parsing and anomaly detected are used, and augmenting them with large language model to provide actionable insights and helpful recommendations.

1.2 Problem Statement

Despite the important role of log analysis that play in making the system more secure and reliable, several significant challenges still persist in current practices:

- *Limited interpretability.* In log anomaly detection, the ability to interpret model's outputs is important for people who work as system administrators or analysts to effectively action to the alerts. They need to understand which log entries may be responsible for the detected abnormality. Yet, many traditional approaches only provide basic classified prediction without any explanation. As a result, engineers still have to perform further manual root cause analysis which in large-scale and complex systems becomes an very time-consuming and heavy task.
- *Poor adaptability.* Many existing methods rely on a predefined set of log event templates during feature extraction phase (This phase will use the set of log event templates generated by log parsing to create numerical features for machine learning models [15]). However, as applications scale up and expand in term of feature, new and unseen log events will definitely appear. Therefore, adapting to these changes require retraining models from scratch which make the systems become less practical in dynamic environments.
- *Poor adaptability.* Many current approaches are using static rules or models trained on old data which make them struggle to adapt to new log patterns or new unseen failures. This lack adaptability reduces the effectiveness of the methods and make it impractical in real-word scenarios.

1.3 Objectives of the Study

This study aim to perform the investigation on existing anomaly-detection methodologies, compares the performance of different machine-learning models, and develops a practical framework that integrates large language models (LLMs) to automate and improve anomaly detection in real-world scenario. At the end of the study, there are some

main objectives that this study aim to achieve:

- (1) Implementing an application for providing a better user interface in anomaly detection fields.
- (2) Applying parsing tools and machine learning algorithms for log anomaly detection.
- (3) Integrate large language models into the system to generate explanations and actionable mitigation recommendations.

1.4 Limitations of the Study

Several limitations are still presented in this research. Firstly, the study focused only on HDFS logs, so therefore the result may not be directly applicable for other types such as application logs or network device logs. Second, the model evaluation was conducted on pre-labeled dataset which may not fully address the challenges that related to real production environments. Finally, scalability testing was not conducted on the system to examine its performance under extensive user loads. The system is designed to handle scalability in mind but the real validation was beyond the scope of this thesis.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction to system log

System logs, also known as event logs or audit trails, are automatically generated by a computer system (its operating system, services, applications) that record events, changes and error that occur inside that system. These log files serve an important role in monitoring system health, diagnose failures, detecting anomalies, and ensure system security and compliance [17]. System logs usually follow a semi-structured format which consist of several fields that capture both the context and content of the event.

```
1 2016-10-02 12:24:52,337 INFO org.apache.hadoop.hdfs.server.datanode.DataNode.clienttrace: src: /10.10.34.11:58237, dest: /10.10.34.11:50010,
   bytes: 144, op: HDFS_WRITE, cliID: DFSClient_NONMAPREDUCE_1903091109_103, offset: 0, srvID: d9ef1b17-4314-4cd8-91eb-095413c3427f,
   blockid: BP-108841162-10.10.34.11-1440074360971:blk_1074555986_815162, duration: 16128279
2 2016-10-02 12:24:52,337 INFO org.apache.hadoop.hdfs.server.datanode.DataNode: PacketResponder:
   BP-108841162-10.10.34.11-1440074360971:blk_1074555986_815162, type=HAS_DOWNSTREAM_IN_PIPELINE terminating
3 2016-10-02 12:24:52,393 INFO org.apache.hadoop.hdfs.server.datanode.DataNode: Receiving
   BP-108841162-10.10.34.11-1440074360971:blk_1074555988_815164 src: /10.10.34.11:58242 dest: /10.10.34.11:50010
4 2016-10-02 12:24:52,394 INFO org.apache.hadoop.hdfs.server.datanode.DataNode: Receiving
   BP-108841162-10.10.34.11-1440074360971:blk_1074555989_815165 src: /10.10.34.11:58243 dest: /10.10.34.11:50010
5 2016-10-02 12:24:52,399 INFO org.apache.hadoop.hdfs.server.datanode.DataNode.clienttrace: src: /10.10.34.11:58242, dest: /10.10.34.11:50010,
   bytes: 66, op: HDFS_WRITE, cliID: DFSClient_NONMAPREDUCE_1530486806_104, offset: 0, srvID: d9ef1b17-4314-4cd8-91eb-095413c3427f,
   blockid: BP-108841162-10.10.34.11-1440074360971:blk_1074555988_815164, duration: 2155456
```

Figure 2.1: An example of raw logs taken from HDFS dataset

Figure 2.1 show a log snippet that was generated by HDFS DataNode, one of the core storage components in the Hadoop ecosystem (a more detail explanation of this system will be provided below). These log entries are created from the storage operations of the system such as data writes, packet handling, and communication between DataNodes. A log entry typically follow a structure as follow:

- *Timestamp:* 2016-10-02 12:24:52,337. Indicates the exact moment when the

system processed the event.

- *Log level:* `INFO`. Shows that the entry reports normal operational activity
- *Log message:* `org.apache.hadoop.hdfs.server.datanode.DataNode.clienttrace:...`
Log messages provide a structured description of system behavior, capturing events, performance indicators. This is the most important part of a log entries and therefore become the foundation for monitoring, auditing and anomaly detection.

Given their important role in providing such essential information about the system, logs must be reliably collected, centrally stored and appropriately maintained to enable accurate monitoring, troubleshooting, and subsequent analytical processes.

2.2 Traditional log analysis method

Traditional log analysis have relied on two primary approaches (cite something): manual log inspection and rule-based pattern matching systems. While these techniques are the foundation of the system administrators fields and troubleshooting, it were used during the time when the system complexity and log volumes were much smaller than nowadays modern systems. Understanding the strengths and limitations of these approaches will provide you a better context and insights of logs which will enhance your log analysis systems.

2.2.1 Manual Log Inspection

Manual log inspection is one of the earliest and most straightforward methods used by system administrators to identify issues. In this approach, developers or administrators will directly operate searching on raw log files using command-line tools such as `grep`, `tail`, `less`, to locate error logs or correlate timestamp of the incidents. While feasible for small log volumes, this method is very time-consuming, highly error-prone and relies heavily on an operator's familiarity with the system. Therefore, this approach is no longer a feasible solution for detecting anomalies in large-scale systems which

may produce millions of log entries per hour. For instance, in 2013 the Alibaba Cloud system was reported to generate approximately 100 to 200 million log lines in one hour [13].

2.2.2 Rule-Based Systems (Regex and Pattern Matching)

To counter the limitations of manual log review, developers created rule-based system that used fixed rules, regular expressions (regex), and predefined patterns to filter, parse, and identify specific events when a specification conditions are met. For example, system operators can create rules to detect keywords such as "ERROR", "FAILED" or extract key information from log entries. Rule-based methods offer some advantages: they are easy to implement and effective for detecting obvious or recurring errors. Many traditional monitoring tools such as Nagios, Splunk (early versions) [16], and Logstash [3], they are heavily depend on popular pattern-matching techniques to filter, categorize, and index log data.

However, the effectiveness of rule-based approaches highly depend on the completeness and accuracy of the hand-crafted predefined rules. It will become more defective in a constantly evolving environments where logs format change frequently and new types of anomalies will appear which do not match the existing patterns. Maintaining large sets of regex rules is also a labor-intensive action and require continuous manual updates on patterns which is unsuitable for adaptive anomaly detection.

2.3 Log parsing

After log collection, raw logs are often semi-structured which contains constant parts and variable parts and may vary among different entries. Log parsing process responsibility is to transform those raw unstructured logs into structured format for downstream analysis. Log parsing techniques will try to identified constant/static fields and dynamic fields. The constant part are often log events, log template and the variables part are often the fields that can change (e.g., IP address, thread name, job ID, message ID). For example in Figure 2.2, a log entry is collected from Hadoop Distributed File System

(HDFS) dataset [22] "Received block-562725280853087685 of size 67108864 from 10.251.91.84" is parsed to log event "Received block <*> of size <*> from <*>", where all changed parameters are captured in "<*>". There are two common parsing techniques: clustering-based (e.g., LogSig [18], LKE[4]) and heuristic-based (e.g., SLCT [19], iPLoM [12]). In the former method, distance between logs is calculate at the beginning, and use clustering techniques to group logs in to cluster. Event template is generated at the end for each cluster. For the heuristic-based approaches, it count the appearance of ech log entries and composed frequent words as event candidate. And finally, some are chosen to become log events.

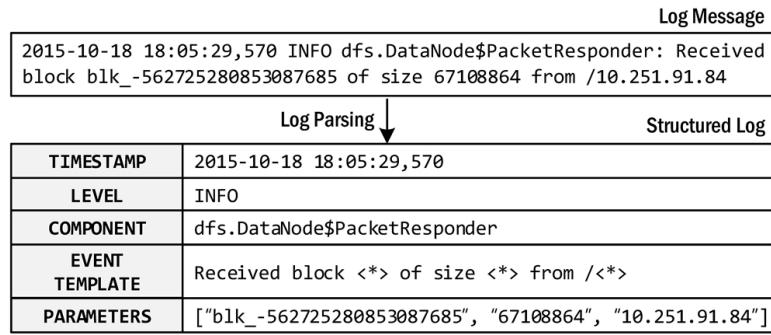


Figure 2.2: Example of Log Parsing

2.4 Machine learning and deep learning for log anomaly detection

As system continuously scaled and logs volumes grew bigger, manual or rule-based approaches is no longer available. As a result, researchers began to adopt machine learning to automate anomaly detection. These methods are created to enable it to learn the structures, numerical representations, and patterns of logs without depending on predefined rules or templates. These approaches are often categorized as supervised or unsupervised learning.

2.4.1 Supervised learning

- a. Support Vector Machine (SVM)

Support Vector Machine is one of the fundamental methods that proposed by Vapnik [20] which is used for binary classification. The basic idea of this method is as follow. For nonlinear separable data, we can transform it to higher-dimensional space where the data becomes linearly separable. This allows the classifier to construct an optimal separating hyperplane between normal and anomalous log patterns. In log detection context, anomaly logs will be treated as negatives and normal one will be served as positive examples. But using this method for detecting anomalies often often suffers from class imbalance, where the positive examples is outnumber the negatives. As the result, the classifier will overfit the anomalous logs, which make it become weak in detecting new occur, unseen logs. Figure 2.3 shows principle behind using SVM to create an optimal boundary between normal and anomalous log database in a feature space.

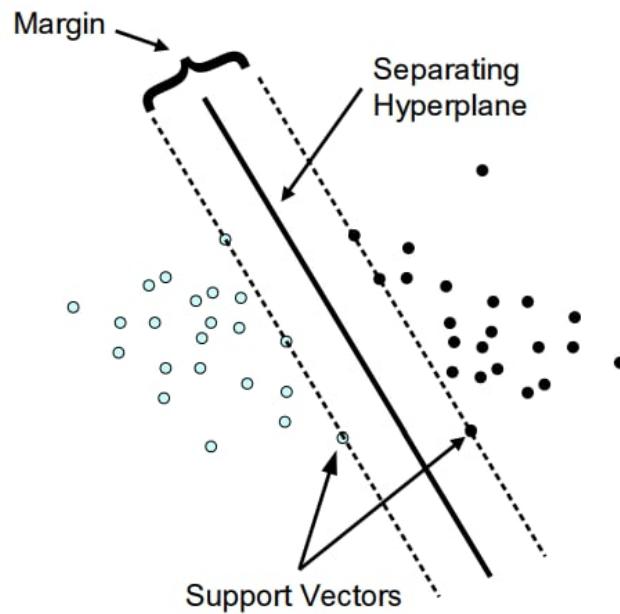


Figure 2.3: SVM classification

b. Decision Tree (DS)

Decision tree is one of the earliest machine learning approaches that apply to anomaly log detection and are defined by Quinlan [14] as "powerful and common tools for classification and prediction". This method recursively splitting

data based on its features then create a tree-like structure where each leaf node represents a prediction class (normal or abnormal). When apply to log anomaly detection, we have to transform log entries into structured numerical features before feed it into decision tree algorithm. The applications of decision trees to log-based anomaly detection has been proposed in many research with promising results. He et al. [6] did some experiments with anomaly logs in HDFS dataset using decision tree classifier and prove the effectiveness of it. The approach not only achieve high accuracy but also provide interpretable rules that allows others to validate and refine them as they wish. In context of this research, decision tree served as a foundation machine learning algorithm for detecting anomalies in log data. The specific implementations and results of using this algorithm will be discussed more upon in subsequent sections.

- c. The final item summarizes the topic.

2.4.2 Unsupervised learning

In many situation when anomaly data is not labeled or unavailable, the supervised learning is thereby become impractical. As a result, unsupervised learning methods have been adopted widely for log anomaly detection. These approaches aim to analyze data without requiring labeled examples by discovering the characteristics of the data itself. This section will examine some popular unsupervised approaches that are currently applied to log analysis.

- a. Isolation Forrest

Isolation Forrest is a machine learning algorithm that was introduced by Liu et al [10] in 2012. His core principle of isolation forrest is that anomalies are more accessible to isolate than anomalous points [11]. The algorithms will construct forrest of random binary trees to separate each data point. Each tree us built by randomly selecting a feature from the dataset and a split value within the feature's range, then repetitive split the data until it reach the maximum depth or the instance is isolated.

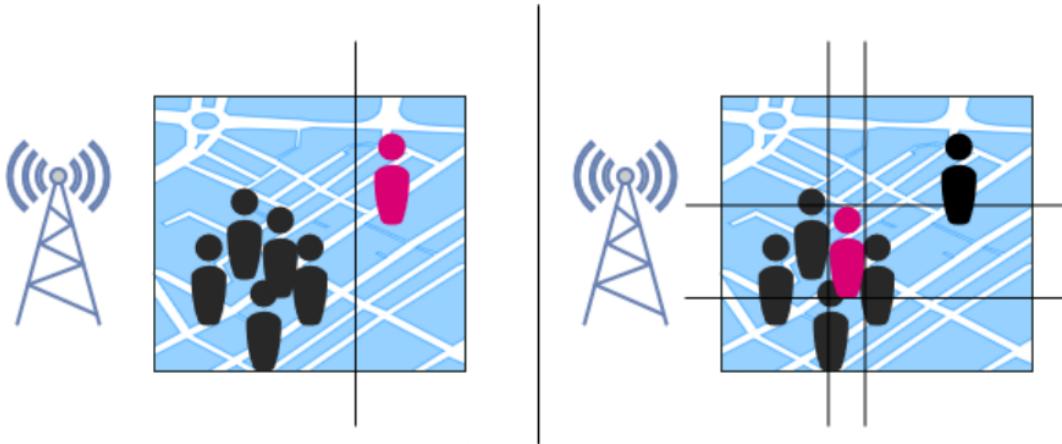


Figure 2.4: Isolation Forest

Figure 2.4 demonstrates the isolation capabilities of Isolation Forest algorithm. In the left, the data point is separated with only one split, indicating a high anomaly score. On the other hand, data point on the right requires more splits, suggesting it's a nominal data point

b. Principal Component Analysis (PCA)

Principal Component Analysis is a common technique that widely used for dimensionality reduction ¹ in a dataset. By doing so, PCA algorithm helps simplifying complex dataset, reducing the redundancy among features, and highlighting the most important patterns in the data [8]. In anomaly detection, it first converts each log sequence into event count vector by counting how many time each type of log event appear in the sequence. Next, the PCA algorithm [9, 1] will capture the main patterns of normal log behavior then project those vectors to a learned space (also known as normal space). However, when a log sequence contains anomaly logs such as missing events or unusual log messages, PCA algorithm will mark it as abnormal behavior by calculating the Square Prediction Error (SPE) [7], which measures the deviation from the learned normal patterns.

¹Dimensional reduction is the process of reducing the number of input variables (features) in a dataset keeping as much essential information as possible.

CHAPTER 3: METHODOLOGIES

3.1 Overview

This chapter describes the methodology that used to design, implement, and evaluate the proposed log anomaly detection framework. The methodology follows a pipeline that begins with data collection and preprocessing, then extract the features and using machine learning techniques to detect anomalies. Large language models (LLMS) are also integrated to address the limitations of the traditional approaches, giving more insights and contextual understanding of detected anomalies. This chapter is organized to present each stage of a framework, from system architecture and how data is prepared to model Implementation and evaluation.

3.1.1 Research Approach

In this study, I adopt a design-based experimental research approach that combines traditional machine learning methods with large language model techniques. This approach will focus on constructing a practical, user-friendly and production-ready log anomaly detection framework by integrating proven traditional methods with LLM-based reasoning to addressing both the technical and operational challenges of large-scale log analysis.

The methodology combines well-established components such as the Drain log parsing algorithms to constructs structured log templates from raw log messages, Decision Tree or Isolation Forrest machine learning models are compared to find the most effective one and implemented it as detection instruments.

For data collection, this study relies primarily on secondary collection methods which uses public and available log datasets like HDFS system logs or BGL logs dataset. There will be no primary data involving human participants are collected. The data collection process includes log ingestion, parsing, feature extraction, anomaly labeling (already prepared by experts), and preparation for model training and testing.

This study use a mixed-method of quantitative and qualitative analysis approaches. Quantitative analysis is utilized to evaluate the performance of detection models using real statistics (e.g., detection rates and error rates) and comparison between models. Furthermore, qualitative analysis is applied to the outputs that is generated by LLMs, focusing on their ability to provide root cause explanations and mitigation suggestions.

Ethics are also considered in this study which relate to data usage and research integrity. All datasets that I used are publicly available so it ensures no personal information is exposed. Proper citation and acknowledgment of datasets, tools, and prior research are also maintained. Additionally, this study is responsible for the LLM-generated outputs by put it as only suggestions rather than automated direct action, and thereby preventing unintended operational impact.

3.1.2 System Pipeline Overview

The proposed framework follows a pipeline from transform raw systems logs into structured one and useful recommendations through a sequence of stages. Figure 3.1 illustrates the overall pipeline.

- (1) The pipeline begins with log preprocessing, where raw logs are parsed using the Drain algorithm. This step will convert raw and unstructured log into sequence of structured events and extract event templates.
- (2) Structured log sequences will be the input for the processing step, where it will be transformed to numerical representations and then fed to machine learning models for anomaly detection.

- (3) The final stage integrates large language models (LLMs) to enhance system interpretability. LLMs component will perform semantic analysis based on detected anomalies and generate human-readable explanations about root causes and useful suggestions.

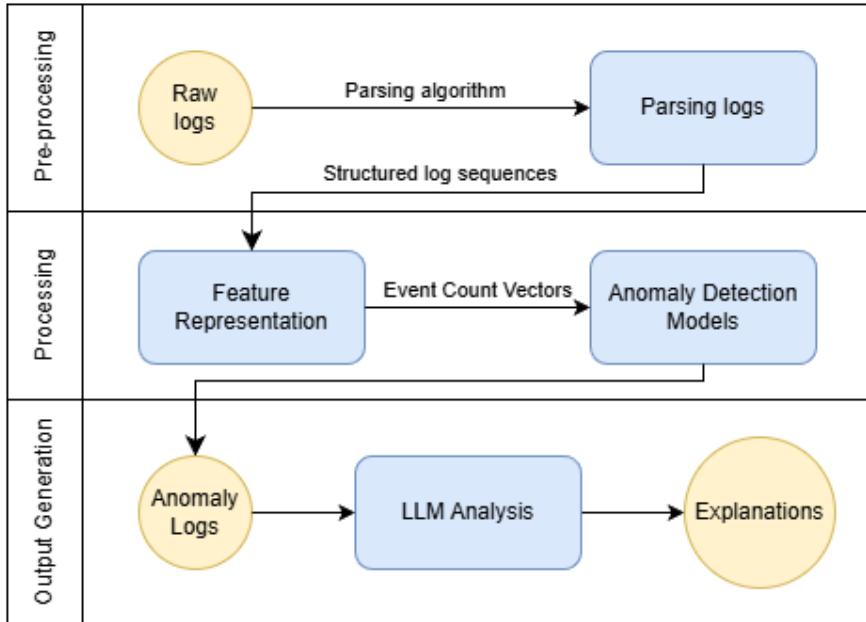


Figure 3.1: System pipeline

3.2 System Architecture and Design

This section presents the system architecture and design of the proposed log anomaly detection framework. We will focus on the structural composition of the system and also show their responsibilities, and interactions. The architecture is designed to make the process of large-scale log data more efficiently while enabling to integrate with traditional anomaly detection techniques and large language models (LLMs).

3.2.1 Architectural Overview and Design Principles

The proposed log analysis system follows a simple architecture but highlight the practicality and ease of integration to support real-world problems. The systems is designed around some key principles:

1. Asynchronous Processing

The architecture apply asynchronous task processing to handle computationally heavy workload without affect user interface. Therefore the system can effective process large log files while also maintain good user interaction.

2. Modular Service Design

Each component in the system is designed as an independent module with clear responsibilities to maintain or test more easily, and also increase the ability of horizontally scaling of individual components.

3. Data Layer Separation

The system create a clear separation between metadata management and object storage that using MinIO for large-scale objects and PostgreSQL for logs structured metadata. This design optimizes the performance by addressing the data access pattern problem where PostgreSQL is suitable for structured queries and indexing whereas MinIO offer high throughput and cost efficiency for actual log storage.

4. Multi-Stage Processing Pipeline

In my log analysis system, I implemented a multi-stage pipeline which consists of parsing, anomaly detection, and intelligent analysis phases. This allows me to test and integrate different algorithms and models that is the most suitable for the system without disrupting the entire workflow.

3.2.2 High-Level System Architecture

The system uses a layered, distributed architecture which is designed to support efficient log ingestion and anomaly detection. Each layer is responsible for a distinct functionality ensuring the ease for development and maintenance. The layered architecture of the system is captured in Figure , and describes in detail as follows.

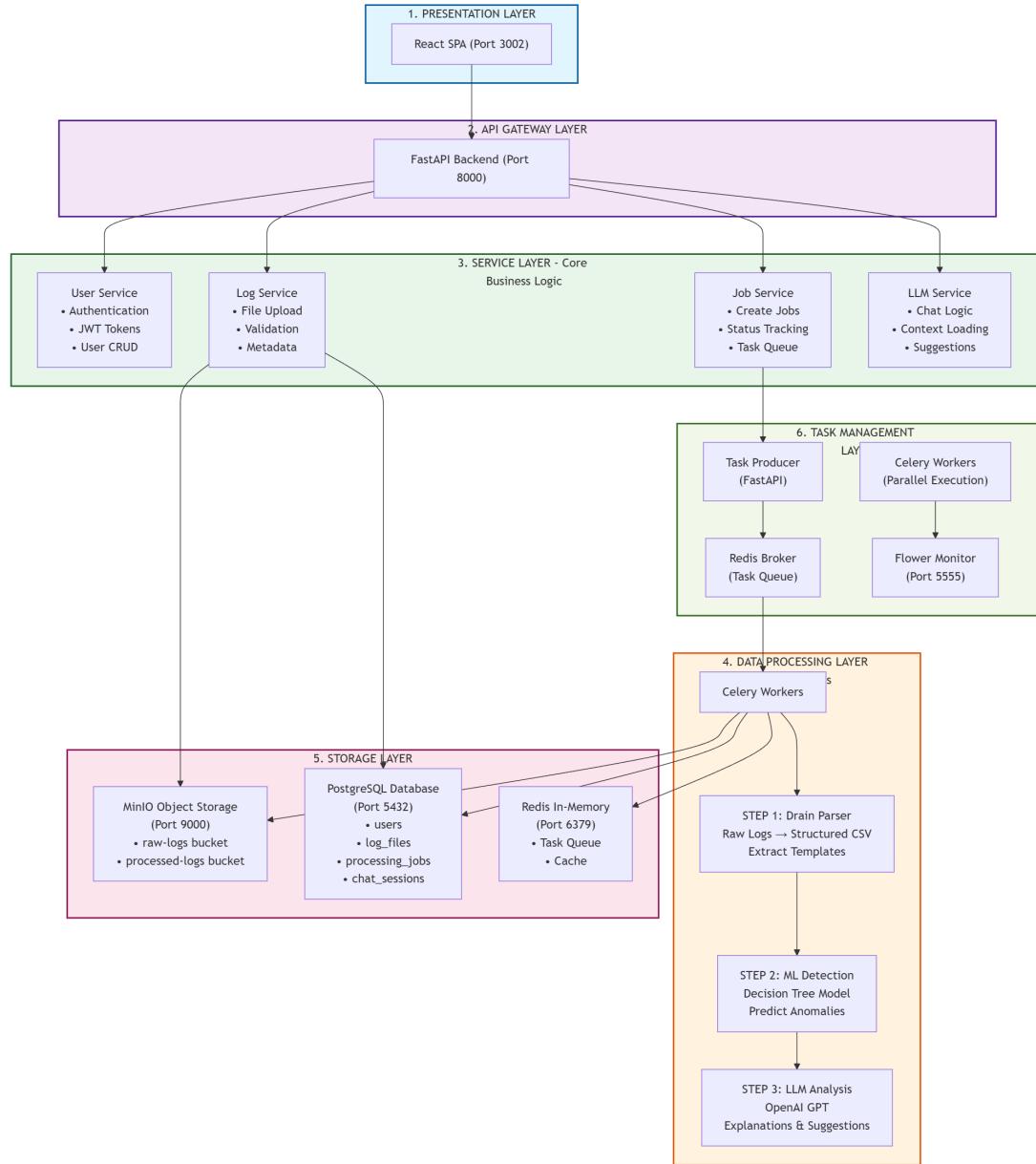


Figure 3.2: System architecture

1. Presentation layer

This layer provides a user-friendly interface between end users and the log analysis system. It is a single-page application (SPA) that built on React framework, allowing users to upload log files, view processing status and final analyzed results. The system also develop authentication through JWT-based mechanism and role-based accesses for better security.

2. API Gateway Layer

The API gateway layer is responsible for creating a communication between the frontend and backend. This layer exposes REST API endpoints which will handle request routing, input validation and give back the responses by using FastAPI technologies.

3. Service Layer

The service layer holds the core business logic of the system. These services include authentication, handling log files, and managing model detection logic, and LLMs analysis. By separating functions into discrete services, the architecture allows each service to work independently and easily to change as you wish.

4. Data Processing Layer

Data processing is one of the most important steps in the entire workflow, so this layer is built very carefully. Raw log messages are parsed into structured ones and extract event templates using Drain algorithm. Then using the structured logs to ingest into machine learning models and save as serialized files for real-time inference. Anomalies are then analyzed by LLMs to generate explanations about how it marks as anomalies and mitigation suggestions.

5. Storage Layer

The storage layer is designed to handle how log data will be stored and accessed effectively. MinIO is used to store actual raw log files that users upload to the server. I utilized this technology because its object storage allows storing and accessing large volumes of data with high speed and low cost. Whereas PostgreSQL is chosen to manage structured metadata due to its ability to create complex structured queries. An in-memory cache (Redis) is also employed in this system to support fast access to data such as task queues.

6. Task Management Layer

To support asynchronous processing as I mentioned in section 3.2.1, the system also create a layer for task management. Each step such as parsing, anomaly detection, and LLM inference will be define as tasks and put to message queues. Celery workers will take tasks from queue and execute in the background, thereby parallel execution is made possible.

3.3 Dataset preparation

The HDFS log data set is the most frequently used data set for anomaly detection methods and thus also the main focus point of this study. The logs were collected from the Hadoop Distributed File System (HDFS), which runs on the Amazon EC2 platform. This system is designed to run on commodity hardware and allows users to store and process large files. Each log event have one or more block identifiers which will enable grouping logs into sequence of events. And in fact, the sample logs snippet show in Figure 2.1 are taken from HDFS daWta set. The core idea behind the anomaly detection in this dataset is that some data blocks is fail to be processed by the system. When such failures happen, the blocks generate log events that differ from normal processing behavior. Therefore, the entire sequence should be detected as anomalous [21].

The data was originally collected by Xu et al. from a production hadoop cluster comprising more than 200 nodes. The dataset was labeled by domain experts to distinguish normal and anomalous execution flows. In this study the HDFS dataset version is taken from project Loghub [22]. The total lines of log messages in this dataset version is 11,175,629 but a close inspection of the dataset shows that approximately 22,000 lines are missing in comparison to the original data set for unknown reasons. However, the remaining data still provides a sufficient sample for anomaly detection analysis and techniques evaluation.

3.4 Log Parsing Methodology

In this section, I briefly introduce to Drain the main parsing techniques used in this study which is a fixed depth tree-based method. The core principle of this algorithm is to organize log messages into a parse tree where each node represents a token position in the log messages [5]. Drain parser first tokenize the log message typically using whitespace as delimiter. For example, log message like "Received block-562725 from node 10.0.0.1" is split into ["Receive", "block", "blk_562725", "from", "node", "10.0.0.1"] and these tokens are used to traverse through layer of a parse tree. The parse tree is illustrated in Figure 3.3. At the first layer under the root node, the tree branches are represented number of tokens in log messages. Messages with the same length are routed to the same subtree. At the next layer , Drain use specific tokens to guide the route. If token have words like "Sends" or "Receive", it is use as branch key. If token is identified as variables part, it will be replace with symbol *.

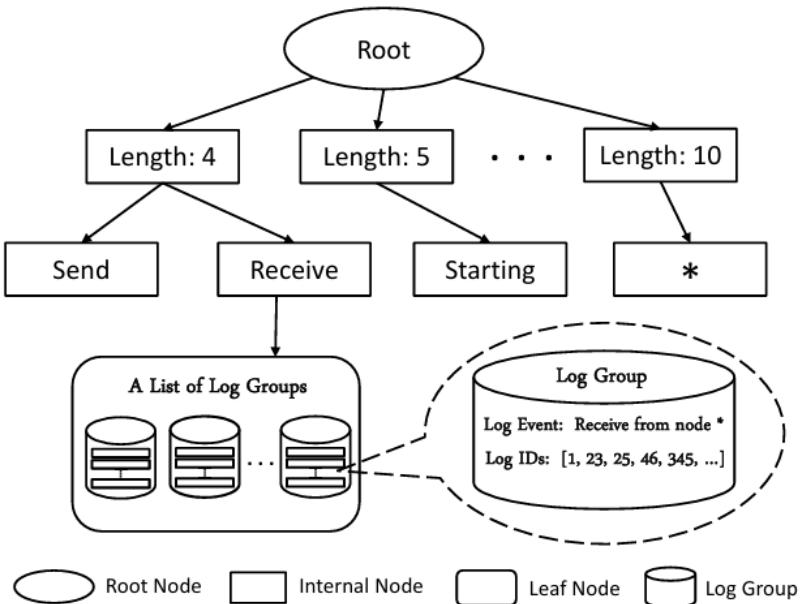


Figure 3.3: Structure of Parse Tree in Drain (depth = 3)

3.5 Feature Extraction / Representation

The main purpose of this step is to extract features that is valuable that can be utilized by anomaly detection models. The input of this step is structured log events generated during the log parsing then the output is event count matrix that represent characteristics of log sequences. The log must be organized in log sequences in order to construct features. This grouping process can be done through windowing techniques. In this project, three windowing techniques are considered.

1. Fixed Window

Fixed windows groups logs based on timestamp using predefined tim duration. All log event that occurred in this duration are grouped in to log sequence. This approach is simple but mat mix unrelated log entry if it occur in the same time window.

2. Sliding Window

Sliding window is the enhance version of fixed window by adding a step size to window size. In general, step size is smaller than window size, therefore overlapping windows is occur.

3. Session Window

Session window is based on identifiers rather than timestamp compare to two other windowing types. This method is practically suitable for distributed system where every log contains an identifiers. As a result, logs with the same identifiers are are grouped into a single session window. In this project, session-based windowing is primarily used for the HDFS dataset.

Once the log sequences are created, feature vectors are generated by counting the occurrence of each log event in a sequence. Each log sequence is represented as an event count vector.

3.6 Anomaly Detection Models

To evaluate and select the most suitable approach for the anomaly detection framework, multiple machine learning models were trained and compared. Supervised machine learning algorithms that used are Decision Tree and Support Vector Machine (SVM), while unsupervised methods are Log Clustering, PCA, Invariants Mining, and Isolation Forest. Each model was trained using the same training dataset to ensure a fair comparison. Performance was evaluated on a test set using precision, recall, and F1-score.

CHAPTER 4: IMPLEMENTATIONS

4.1 Implementation Overview

This chapter will present the implementation of the proposed log anomaly detection application. Each components that are described in chapter 3 will be transformed to real functional code and integrate in to the system. The scope of this chapter is focus on implementation details including configuration, library used, API designs and technologies that are used to build end-to-end system.

4.2 Implementation timeline

The system implementation is followed a structured timeline that consists of several phases and each phase will focus on a specific tasks and will be described as follow:

- *Phase 1 (Week 1)*

This initial phase focused on prepare and research background knowledge for the project. Some major activities are:

- Creating a literature review about log parsing, anomaly detection techniques, and LLMs for log analysis.
- Understanding about HDFS dataset, how it was generated, and its structure.
- Defining system requirements and architecture.

- *Phase 2 (Week 2): Data Preparation*

In this phase, raw actual log will be collected, and prepared for use in subsequent steps.

- Downloading raw HDFS dataset from LogHub repository and validating data.
 - Using a small sample in the data set to understand the event structure.
 - Set up a directories just for data storage purposes.
- *Phase 3 (Week 3-5): Backend Infrastructure Setup*

The fundamental backend environment was set up carefully for later implementation:

- Setting up the FastAPI project structure, base routers, and middleware.
 - Configuring PostgreSQL database, Redis, and MinIO containers using Docker.
 - Designing database schema.
- *Phase 4 (Week 6-7): Log Parsing module development*

This phase mainly focused on implementing a wrapper for Drain parser in order to integrate into the system, which includes some major steps:

- Developing a Python script to wrap the original Drain algorithm.
 - Creating Celery tasks for asynchronous parsing.
 - Designing storage solution for parsed logs (MinIO + PostgreSQL)
- *Phase 5 (Week 8-9): Log Parsing module development*

Some popular ML anomaly detection models were utilized and evaluated in this phase:

- Extracting event count vectors and features.
- Training and validating models using parsed event sequences.

- Storing trained models for reusing in the processing pipeline.
- *Phase 6 (Week 10-11): LLM Integration*

This phase focused on implementing LLM for more understanding of anomalies:

 - Developing the LLM service module for interacting with the model API.
 - Structuring a prompt templates for explanation, and mitigation suggestions.
 - Integrating into the pipeline as the final stage.

- *Phase 7 (Week 12-13): Frontend Development*

A functional web interface is constructed and delivered in this phase using React technology:

 - Developing the LLM service module for interacting with the model API.
 - Structuring a prompt templates for explanation, and mitigation suggestions.
 - Integrating into the pipeline as the final stage.

- *Phase 8 (Week 14): Testing, Optimization, and Debugging*

This phase performs several tests and optimization to ensure product stability:

 - Optimizing database queries and model loading.
 - Refining UI responsiveness and user experience flow.
 - Running test with a full datasets to ensure all stages work smoothly.

- *Phase 9 (Week 15-16): Documentation and Thesis writing*

The final phase will consolidate all the work and prepare the system for thesis writing:

- Documenting code structure, APIs, preprocessing steps, and models.
- Preparing visualizations, diagrams, and comparison tables.

- Create overall structure for thesis and complete the final work.

4.3 Backend Implementation

The backend of the log analysis system was developed using FastAPI - a popular Python framework , due to its ease of use and high performance. The implementation follows the layered architecture that introduced in Chapter 3 and therefore separate the entire system into modular routers and components in order to maintain and scale more easily.

4.3.1 FastAPI Application Architecture

a. Application Initialization

The main application entry point is defined in ‘src/main.py‘ as shown in Figure 4.1, where all routers and middlewares are registered. The main application is also responsible for initializing CORS middleware and create all database tables that defined in SQLAlchemy models whenever the application start. This setup system features like Auth, Jobs Logs, to be managed by it own route files.

```

1 app = FastAPI()
2
3 # Add CORS middleware
4 app.add_middleware(
5     CORSMiddleware,
6     allow_origins=["*"],
7     allow_credentials=True,
8     allow_methods=["*"],
9     allow_headers=["*"],
10 )
11
12 app.include_router(auth_router.router)
13 app.include_router(logs_router.router)
14 app.include_router(jobs_router.router)
15 app.include_router(llm_router.router)
16
17 Base.metadata.create_all(engine)
18

```

Figure 4.1: main.py file

b. Router Organization

Router	Responsibilities
auth_router	Login, registration, token generation
log_router	File upload, metadata retrieval
job_router	Job creation, status queries, result retrieval
ml_router	Model inference endpoints
llm_router	Explanation and mitigation generation

Table 4.1: Router Responsibilities

Each domain in the system is separated to different router to maintain the ability to scale, be readable when number of features and endpoint grows. The table 4.1 gives information about the total router that are implemented in the systems and their responsibilities. Routers itself will only communicate with service layers to maintain a well and clear application architecture.

```

1 router = APIRouter(prefix="/auth", tags=["Authentication"])
2
3 # Get user login
4 @router.get('/user', status_code=status.HTTP_200_OK)
5 def get_user_login(current_user: CurrentUser, db: Session = Depends(get_db)):
6     user_service = UserService(db)
7     user_login = user_service.get_username(current_user)
8     return user_login
9
10 # Get all users
11 @router.get('/admin/users', status_code=status.HTTP_200_OK,
12             response_model=List[schemas.UserResponse], summary="Get all users")
13
14 def get_all_user(current_admin: AdminUser, db: Session = Depends(get_db)):
15     user_service = UserService(db)
16     users = user_service.get_all_users()
17     return users
18
19
20 @router.post('/users', status_code=status.HTTP_201_CREATED,
21             response_model=schemas.User, summary="Create new user")
22
23 def create_user(request: schemas.User, db: Session = Depends(get_db)):
24     user_service = UserService(db)
25     new_user = user_service.create_user(request)
26     return new_user

```

Figure 4.2: Example of a router configuration pattern

Figure 4.2 captures a code snippet that represent a pattern used for implementing router configuration. Each router will contain several endpoints that directly related to that functional service layer. For example, the authentication router includes endpoints for user creation, get login users or send access tokens. This approach allows developers to organize the application better and avoid as much as confusion in source code.

4.3.2 Database Layer Implementation

fSupport Vector Machine (SVM) The database layer was built based on PostgreSQL database with SQLAlchemy ORM which allows developers to interact with relational databases using Python classes and objects instead of using raw SQL queries. All entities in the system such as users, logs, jobs and analysis are represented as ORM models

ann then mapped to relational database tables.

a. Database schema

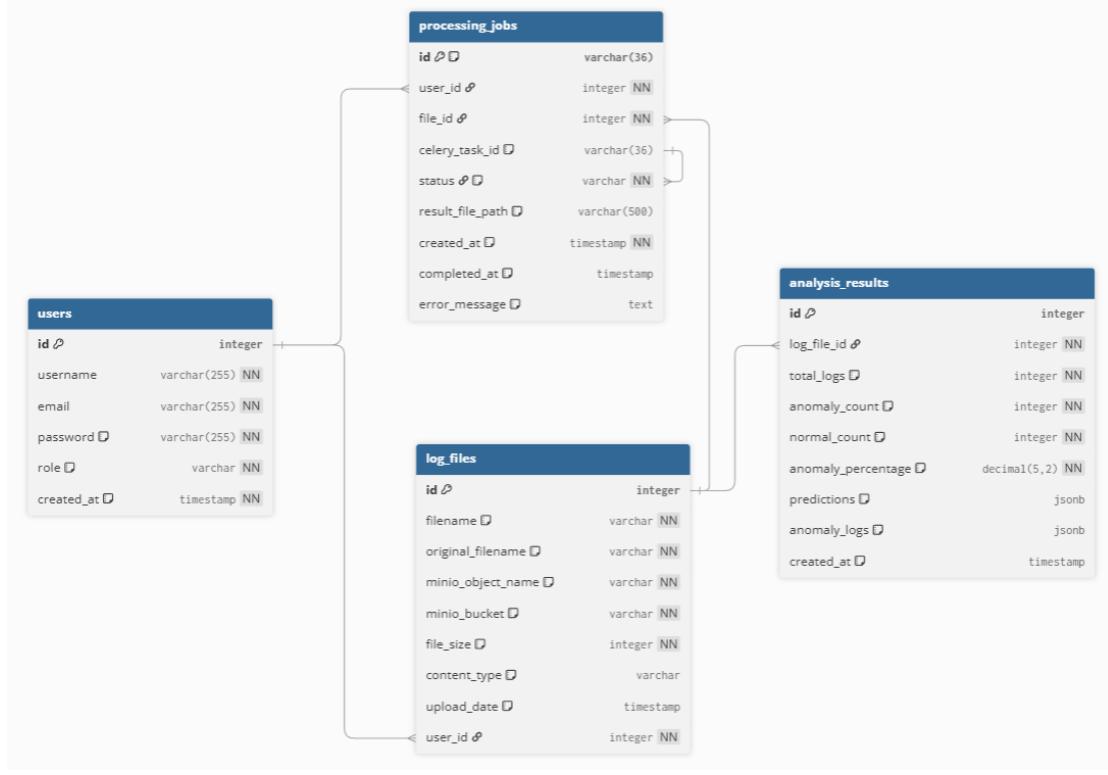


Figure 4.3: System database schema

Figure 4.3 depicts 4 core tables in the anomaly detection system, including `users`, `log_files`, `processing_jobs` and `analysis_results`. Each table serves a distinct purpose and are described as follow:

- Table `users`: This table will manage users information such as user name, password, email address and their role in the system.
- Table `log_files`: This table stores metadata of a log file that uploaded by users, and there will be a link to its actual storage location in MinIO.
- Table `processing_jobs`: This table is primarily used for storing jobs information that create by the system to track the Celery background tasks.
- Table `analysis_results`: ML anomaly detection results for each analyzed log

filed are stored in this table with some information such as number of sequences, anomaly and normal sequences count

The database schema also implement logical relationships between entities. Each user can upload multiple log files and create multiple processing jobs. The log file entity is a central reference point to connect both processing jobs and analysis results. This implement means each log file can be associated with multiple processing jobs, which happen when reprocessing is required due to any failures in the pipeline. In addition, each log file can also link to multiple analysis results to support the LLM integration stage.

b. Model implementation

```
● ● ●  
1  class Users(Base):  
2      __tablename__ = 'users'  
3      id = Column(Integer, primary_key=True, index=True)  
4      username = Column(String(255), unique=True)  
5      email = Column(String(255))  
6      password = Column(String(255))  
7      role = Column(Enum(UserRole), default=UserRole.user, nullable=False)  
8      created_at = Column(DateTime(timezone=True), server_default=func.now(), nullable=False)  
9  
10     log_files = relationship("LogFile", back_populates="owner")
```

Figure 4.4: System database schema

In this project SQLAlchemy ORM (Object Relational Mapping) is used to create a clean abstraction over a complex relational database. There are will be four model configuration file for four entities coordinatly. Each file is defined how information of a entity is store and connected within our database. Figure 4.4 shows how a model (Users model) is initialize in the project. The Users entity stores authentication information with server key attributes such as id which is identified for each user, username and password to log into the system. This file also so handle relationship with other entities, in this case, Users model have a one-to-many relationship with the LogFile entity. Each user can upload multiple log files, and each log file link to its owner via a foreign key.

4.3.3 API Endpoints Implementation

Table 4.2 described all the endpoints that are exposed by the backend of the system that cover authentication, log management, job processing, and LLM results generation. Each endpoint is responsible for a specific functions in the entire processing pipeline.

	Method	Endpoint	Auth Required	Description
Authentication Endpoints (/auth)	POST	/auth/token	No	Login and get JWT access token
	POST	/auth/users	No	Register new user account
	GET	/auth/user	Yes (User)	Get current authenticated user info
	PUT	/auth/change-password	Yes (User)	Change user password
	GET	/auth/admin/users	Yes (Admin)	Get all users in system
Log Management Endpoints (/api/logs)	POST	/api/logs/upload	Yes (User)	Upload log file
	GET	/api/logs/logs	Yes (Admin)	Get all log files metadata
	GET	/api/logs/logs-size	Yes (Admin)	Calculate total storage size
	GET	/api/logs/uploads-by-date	Yes (Admin)	Get upload statistics by date (query param: days)
Job Processing Endpoints (/api/jobs)	GET	/api/jobs/recent	Yes (User)	Get user's recent processing jobs (query param: limit)
	GET	/api/jobs/{job_id}/status	Yes (User)	Get current job status and progress
	GET	/api/jobs/{job_id}/stream	Yes (User)	Server-Sent Events for real-time progress updates
	GET	/api/jobs/{job_id}/results	Yes (User)	Get ML analysis results for completed job
LLM Analysis Endpoints (/api/llm)	GET	/api/llm/analyze-anomaly	Yes (User)	Analyze anomaly log with LLM using session context

Table 4.2: API endpoints

Endpoints implementation detail:

This section will provide detail of technical implementation of some important API endpoints.

1. POST /api/logs/upload

This is the core endpoint that handles the file upload process in the system workflow. It exposes a interface for users to upload their log file, store them in object storage system, register a record in the dataset and finally trigger an asynchronous processing jobs for further analysis. Detail code implement is captured in Figure 4.5 and described as follow.

The endpoint is implemented as HTTP POST route under path /api/logs/upload. It is defined using FastAPI's router and use asynchronous mechanism to handle request in order to ensure non-blocking I/O operations. When the backend receive a request from this endpoint, it first initialize a LogService instance which is a service layer that hold the business logic. The uploaded file is then validated for allowed extension. If the file extension does not match the .log extension, the method will immediately return HTTPException with status code 400 which is indicated a bad request error. The endpoint is continue to operate LogService.save_log_file() method which is responsible for uploading valid log files to MinIO and save its metadata to the database. Finally, the method return a structured results of several information.



```
1  @router.post("/upload", response_model=schemas.UploadResponse)
2  async def upload_log_file(current_user: CurrentUser,
3                           file: UploadFile = File(...),
4                           db: Session = Depends(get_db)):
5
6      log_service = LogService(db)
7      allowed_extensions = [".log"]
8      file_extension = os.path.splitext(file.filename)[1].lower()
9
10     if file_extension not in allowed_extensions:
11         raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
12                             detail=f"File type not allowed.")
13
14     result = log_service.save_log_file(file, current_user["id"])
15
16     if result["error"]:
17         return schemas.UploadResponseFail(message=result["message"], error=True)
18
19     # Extract job_id from job_data
20     job_id = result.get("job_data", {}).get("job_id")
21
22     return schemas.UploadResponseSuccess(
23         file_id=result["file_id"],
24         bucket=result["bucket"],
25         object_name=result["object_name"],
26         message=result["message"],
27         user_id=result["user_id"],
28         job_id=job_id, # Add job_id to response
29         error=False
30     )
```

Figure 4.5: Logs upload endpoints

2. POST /api/auth/token

Figure 4.6 describes a simple implementation of token processing endpoint. This endpoint is responsible for handling JSON Web Tokens (JWTs) to verify user within the system. Once the backend receive request, UserService will perform user finding, password verification and generate token. When the authentication is successfully done, the service will create JWT which contains user's information and attach it to request header when accessing protected resources.

```
● ● ●  
1  @router.post('/token', response_model=schemas.Token)  
2  def login_for_token(form_data: Annotated[OAuth2PasswordRequestForm, Depends()],  
3                      db: Session = Depends(get_db)):  
4      auth_service = UserService(db)  
5      token_data = auth_service.login_for_access_token(form_data)  
6      return token_data  
7
```

Figure 4.6: Token endpoints

4.4 Log Processing Pipeline

This section provides detail implementations of multi-stage log processing pipeline within the system. This pipeline is responsible for transform raw logs into structured data and perform anomaly detection with machine learning algorithm.

4.4.1 File Upload and Storage

a. MinIO Client Integration

MinIO was selected as the object storage solution in this project to store raw log data due to its high performance and suitable for containerized deployments. The system implements a Python MinIOClient wrapper class that abstracts from the MinIO Python SDK. Figure 4.7 captures the code snippet of MinIOClient class from the system.

```

● ○ ●

 1  class MinIOClient:
 2      def __init__(self):
 3          self.client = Minio(
 4              settings.MINIO_ENDPOINT,
 5              access_key=settings.MINIO_ACCESS_KEY,
 6              secret_key=settings.MINIO_SECRET_KEY,
 7              secure=settings.MINIO_SECURE
 8          )
 9          self._ensure_bucket_exists()
10
11     def _ensure_bucket_exists(self):
12         """Create bucket if it doesn't exist"""
13         try:
14             if not self.client.bucket_exists(settings.MINIO_BUCKET_NAME):
15                 self.client.make_bucket(settings.MINIO_BUCKET_NAME)
16                 print(f"Bucket '{settings.MINIO_BUCKET_NAME}' created")
17         except S3Error as e:
18             print(f"Error creating bucket: {e}")
19
20     ...
21
22     minio_client = MinIOClient()

```

Figure 4.7: minio_client.py file

When initializing this class, the client creates a connection using predefined access credentials and also verifies bucket with `_ensure_bucket_exists()` method. This ensure the target bucket exist and automatically create it if necessary. The application also implement singleton pattern for MinIO client, where only one instance of `MinIOClient` is created and shared across the entire backend. This design allows system to maintain consistent connection with MinIO that helps uploading and storing operations faster.

b. File Upload Implementation

File upload mechanism is implemented inside `upload_file()` method and are captured in Figure 4.8. The method first wrap log file data in `io.BytesIO` to transform data to file-like interface to that required by MinIO SDK. The the method perform an upload to existing MinIO bucket and a return a response with useful information when the upload is done.

```

1  def upload_file(self, file_data: bytes, object_name: str, content_type: str):
2      """Upload file to MinIO"""
3      try:
4          file_stream = io.BytesIO(file_data)
5          file_size = len(file_data)
6
7          self.client.put_object(
8              bucket_name=settings.MINIO_BUCKET_NAME,
9              object_name=object_name,
10             data=file_stream,
11             length=file_size,
12             content_type=content_type
13         )
14
15     return {
16         "bucket": settings.MINIO_BUCKET_NAME,
17         "object_name": object_name,
18         "url": f"{settings.MINIO_ENDPOINT}/{settings.MINIO_BUCKET_NAME}/{object_name}",
19         "message": "File upload successfully"
20     }
21     except S3Error as e:
22         raise Exception(f"MinIO upload error: {e}")

```

Figure 4.8: File upload method

c. Unique Filename Generation

To ensure files that uploaded to MinIO have the unique object name, the system develop a mechanism to create unique name for each object by combining timestamp, random characters and its original filename. As a result, each log file save to MinIO bucket will have result path as logs/user_id/timestamp_uuid_filename.log. This strategy ensure the object name collision probability is extreme low and therefore reduce the error rate within the system. In addition, objects is store in separate user-scope directories which support better audition and efficient lookup.

<input type="checkbox"/>	20251114_051906_9d2b1e17_HDFS_2k.log	Fri, Nov 14 2025 12:19 (GMT+7)	281.1 Kib
<input type="checkbox"/>	20251114_052123_4eb04afe_HDFS_100K.log	Fri, Nov 14 2025 12:21 (GMT+7)	13.4 MiB
<input type="checkbox"/>	20251114_054151_39e8afae_10000_random.log	Fri, Nov 14 2025 12:41 (GMT+7)	1.3 MiB

Figure 4.9: Objects stored in MinIO

4.4.2 Asynchronous Processing with Celery

a. Introduction to distributed task processing

Distributed task processing has become increasingly important in modern system because this technology allows all tasks are distributed efficiently across multiple nodes. This approach enhances system scalability, performance, and enable extensive computation can be efficiently execute. Several tools and framework have been proposed to handle distributed task execution such as Apache Hadoop or Apache Spark. However, in this project, Celery is use to handle distributed task processing which is a framework that built on top of the Advanced Message Queuing Protocol (AMQP) paradigm. For the context of log anomaly detection, for example, parsing a 50MB HDFS log file with traditional subsequent methods require 30-100 seconds to finish, this means the entire system have to wait until the process complete. This issue exceed typical HTTP timeout thresholds, therefore degrading user experience and increase the risk of resource exhaustion. With Celery, instead of waiting for the entire processing to finish, the API only submits its task to a queue and immediately returns control to the user. Celery workers then fetch tasks from the message broker, execute them in the background, and update the job status in the database.

b. Celery Implementation and Configuration

This system employs Celery as asynchronous task processing solution and also integrate with Redis as its message broker. This approach is provides a easier and more lightweight solution for task distribution without operational overhead of another message broker is RabbitMQ.



```
1 celery = Celery(__name__)
2 celery.conf.broker_url = os.environ.get("CELERY_BROKER_URL")
3 celery.conf.result_backend = os.environ.get("CELERY_RESULT_BACKEND")
```

Figure 4.10: Celery application configuration

Figure 4.10 presents a simple configuration to establish Celery application and the connection to Redis message broker which also acts as a place to store task results. The architecture follows a standard producer-consumer model: when a user uploads a log file through REST API, our FastAPI backend (producer) stores the file in MinIO and then publishes a message(task) to Redis queue with a unique ID. Celery workers running in background, continuously poll the queue for pending tasks. When receiving a task message, a worker executes the processing logic, updates job status in the database, and optionally publishes results back to Redis.

c. Task Implementation

1. Log Processing Task

The system implements a two-stage processing pipeline through chained Celery tasks. The first task is `preprocess_task` which is used for log parsing and transform to structured CSV file. The detailed steps will be described as follows. The Celery worker first retrieves the log file from MinIO object storage and decodes UTF8 content, then invokes the Drain parser service for template extraction. The parser result is then transformed into CSV format to provide human-readable format and serve as input to machine learning task. The structured CSV is uploaded to other MinIO bucket and saved the bucket information to database for retrieving during ML task. Finally, it enqueues the next `ml_analysis_task` to the message queue, this allows the current worker to complete and accept new tasks while letting other workers to process ML tasks. All the implementations as code is shown at Figure 4.11.

```

● ● ●
1  @celery.task(name='create_task', bind=True, base=DatabaseTask)
2  def create_task(self, job_id: str):
3      db = self.db
4      job = None
5
6      try:
7          # Fetch the job and update its status
8          job = db.query(ProcessingJob).filter(ProcessingJob.id == job_id).first()
9          job.status = JobStatus.PROCESSING
10         job.celery_task_id = self.request.id
11         db.commit()
12
13         # Fetch the log file information
14         log_file = db.query(LogFile).filter(LogFile.id == job.file_id).first()
15
16         # Download log file content from minio
17         file_data = minio_client.get_file_raw(log_file.minio_object_name)
18         log_content = file_data.decode('utf-8', errors='ignore')
19
20         # Parse the log content
21         parse_result = parse_log_content(log_content=log_content, log_format_name='hdfs')
22
23         # Convert parsed logs to a structured DataFrame
24         df_structured = pd.DataFrame(parse_result['structured_logs'])
25         df_structured = df_structured.set_index('ID', drop=False) # Assuming 'ID' is the column name for the index-like behavior
26
27         # Define the path for the processed CSV file
28         object_path = f'processed/{job.user_id}/{log_file.filename}_structured.csv'
29
30         # Upload the structured log data to minio as a CSV file
31         minio_client.upload_process_file(file_data=df_structured.to_csv().encode('utf-8'), object_name=object_path, bucket_name='processed-logs')
32
33         # Update job with the path to the result file
34         job.result_file_path = object_path
35         db.commit()
36
37         # Trigger the next analysis task
38         ml_analysis_task.delay(job_id) # Queue ML task
39
40     except Exception as e:
41         if job:
42             # Update job status to failed if an exception occurs
43             job.status = JobStatus.FAILED
44             db.commit()
45         raise

```

Figure 4.11: Log process task implementation

2. Machine Learning Analysis Task

This task is responsible for the second stage of the task chain, where it consume CSV input, perform machine learning detection and present output. At the beginning of the task, worker retrieve structured log CSV file from MinIO, the pre-trained model is then invoked for prediction. After the prediction successfully done, all the important information such as total logs, anomaly count, and prediction log sequences are stored in to database. Only after the analysis task complete, the entire job will be marked as COMPLETE.

4.4.3 Drain Parser Integration

The system encapsulates the Drain parser algorithm that introduced in previous chapter within a custom python class. The approach wrap the entire original complex Drain algorithm into a clean interface for system to interact with. At the beginning the wrapper defines a format presets for common log types. In the current implementation, the HDFS log format is defined using a fixed template that explicitly separates timestamps, process identifiers, log levels, components, and message content. This structured definition ensures consistent field extraction and reduces ambiguity during parsing. When user invoke the the parser service with a specific log format, these optimized settings wil be applied automatically. In Figure 4.12, the implementation will provide a constructor for `DrainParserService` with several parameters such as similarity threshold (`st`), parse tree depth (`depth`), and maximum children per node (`max_child`). These parameters allow control over parsing behavior which will enable user to customize as they wish. Overall, this approach provides an easy solution and reuseable interface for system to integrate complex algorithm in anywhere.

```

1  class DrainParserService:
2      """
3          Wrapper service for Drain log parser that works with in-memory content
4      """
5
6      # Common log format templates
7      LOG_FORMATS = {
8          "hdfs": "<Date> <Time> <Pid> <Level> <Component>: <Content>"
9      }
10
11     # Common preprocessing regex patterns
12     DEFAULT_REGEX_PATTERNS = {
13         "hdfs": [
14             r'blk_(|-)[0-9]+', # block id
15             r'(/|)([0-9]+\.)\{3}[0-9]+(:[0-9]+|)(:|)', # IP
16             r'(?<=[^A-Za-z0-9])(\|-?\+?\d+)(?=[^A-Za-z0-9])|[0-9]+$', # Numbers
17         ]
18     }
19
20     def __init__(
21         self,
22         log_format: str = None,
23         log_format_name: str = "hdfs",
24         st: float = 0.5,
25         depth: int = 4,
26         max_child: int = 100,
27         rex: List[str] = None,
28         keep_para: bool = True
29     ):

```

Figure 4.12: Drain algorithm wrapper implementation

4.4.4 Feature Extraction

The next step in the data preprocessing pipeline is feature extraction which transform Drain-parsed data in to numerical feature vectors. This sections presents the implementation of integrating Loglizer library's `FeatureExtractor` class to convert event sequences into fixed-dimensional count matrices suitable for machine learning classification. In the proposed system, feature extraction is applied at two distinct phases: (1) During training phase, where model learned and serialized for reuse, (2) during online inference, where the same transformation logic is applied to user lof data.

a. Training Phase

Figure 4.13 shows the configuration for feature extraction during the offline training process. Parse log is first loaded using session-based windowing strategy.

This strategy group structured log events by session identifiers and will mark anomaly label at sequence level not entry level. Once the sequences are prepared, FeatureExtractor class is invoked to convert event sequences into numerical feature vectors that required by machine learning algorithms. This transform is based of event frequency statistics, which mean that whe count the occurrence the number of single log event to form the event count vector. Finally every count vector are formed an event count matrix X , where entry $X_{i,j}$ presents many times the event j occurred in the i -th log sequence [6].

```

● ● ●
1  if __name__ == '__main__':
2      (x_train, y_train), (x_test, y_test) = dataloader.load_HDFS(struct_log,
3                                         label_file=label_file,
4                                         window='session',
5                                         train_ratio=0.5,
6                                         split_type='uniform')
7
8      feature_extractor = preprocessing.FeatureExtractor()
9      x_train = feature_extractor.fit_transform(x_train, term_weighting='tf-idf')
10     x_test = feature_extractor.transform(x_test)
11
12     model = DecisionTree()
13     model.fit(x_train, y_train)

```

Figure 4.13: Feature extraction implementation

b. Inference Phase

In the inference phase, feature extraction is executed inside Celery background task. Parsed log sequences generate from new logs are transform using pre-trained feature extractor. This ensures that the same event vocabulary and weighting scheme learned during training are are consistently applied. This approach helps system avoids feature drift and maintains reliable outcomes.

4.4.5 Machine Learning Implementation for Anomaly Detection

This section will evaluate multiple machine learning models for logs anomaly detection to find the suitable approach that have the most suitable in term of performance and

system efficiency for the project. After log parsing and event sequences construction, Several machine learning models were trained and evaluated on the same dataset to ensure fair comparison. The base code for machine learning algorithms were provided by Loglizer project [6] as mentioned earlier in this study. Each algorithm is training with default parameters to establish baseline performance before hyperparameter optimization. The training pattern followed a consistent flow across all models as showing in Figure 4.14.

```

● ● ●

1  struct_log = '../data/HDFS/HDFS_100k.log_structured.csv' # The structured log file
2  label_file = '../data/HDFS/anomaly_label.csv' # The anomaly label file
3
4  if __name__ == '__main__':
5      (x_train, y_train), (x_test, y_test) = dataloader.load_HDFS(struct_log,
6                           label_file=label_file,
7                           window='session',
8                           train_ratio=0.5,
9                           split_type='uniform')
10
11     feature_extractor = preprocessing.FeatureExtractor()
12     x_train = feature_extractor.fit_transform(x_train, term_weighting='tf-idf')
13     x_test = feature_extractor.transform(x_test)
14
15     model = DecisionTree()
16     model.fit(x_train, y_train)
17
18     print('Train validation:')
19     precision, recall, f1 = model.evaluate(x_train, y_train)
20
21     print('Test validation:')
22     precision, recall, f1 = model.evaluate(x_test, y_test)

```

Figure 4.14: Standard training workflow

Finally, anomaly detection results are stored in the database for further analysis and retrieval.

4.5 LLM Integration

Instead of stopping at machine learning detection phase of some previous study, my system extends the ability to detect anomaly with Large Language Model (LLM). This approach will provide human-readable explanations and recommendations for marked

anomalies.

4.5.1 API client setup and Context preparation

1. API client setup

In this study, Groq's cloud-based inference platform is used to provides free hosting Meta's LLaMA 3.3 model. This approach will avoid the computation overhead and the complexity of the system. Figure 4.15 create a API connection through HTTP client. The service configuration includes 3 keys components: (1) API key that used to connect with Groq service, (2) Groq API endpoint URL which specifically targeting its Chat Completion service, and finally (3) the choice of LLaMA 3.3 model that responsible for generating responses. This model is selected because its strong balance between the capability for reasoning and contextual understanding due to its large parameter size (70 billions).

```
 1  class LLMService:
 2      """LLM service with session context for better analysis."""
 3
 4      def __init__(self, db: Session = None):
 5          self.api_key = os.getenv("GROQ_API_KEY")
 6          if not self.api_key:
 7              raise ValueError("GROQ_API_KEY not found in environment")
 8          self.base_url = "https://api.groq.com/openai/v1/chat/completions"
 9          self.model = "llama-3.3-70b-versatile"
10          self.db = db # Database session for fetching context
```

Figure 4.15: LLM configurations

2. Context Preparation and Session Retrieval

To effectively detect anomaly we can not just examine a single log entry, it is depending on the surrounding logs in the same session. To create this context for LLM, I implement the system so that it can retrieve all log entries that sharing the same anomaly's BlockId from the processed CSV file. This process involves

four steps which is shown in Figure 4.16. First, the system queries the database to find the file path of CSV file in MinIO. Next, the CSV is downloaded from MinIO using chunk approach to prevent unnecessary memory usage. The parsed log entries are then filtered to keep the one that match the target BlockId. Finally, the filtered logs are sorted chronologically based on their LineId to ensure the original execution order.

```

● ○ ●

1  async def _fetch_session_context(self, job_id: str, block_id: str) -> List[Dict]:
2      # Query database for CSV location
3      job = self.db.query(ProcessingJob).filter(
4          ProcessingJob.id == job_id
5      ).first()
6
7      # Download CSV from MinIO
8      csv_data = self.minio_client.get_object(
9          bucket_name="processed-logs",
10         object_name=job.result_file_path
11     )
12
13     # Parse CSV and filter by BlockId
14     df = pd.read_csv(temp_csv.name)
15     session_logs = df[df['BlockId'] == block_id]
16     session_logs = session_logs.sort_values('LineId') # Chronological order
17
18     # Return first 20 logs
19     context = session_logs.head(20).to_dict('records')
20
21     return context

```

Figure 4.16: LLM context preparation

4.5.2 Prompt Engineering

Prompt engineers plays an important role on how our Large Language Model understand the context and response. In this system, prompts are designed to guide the model specific for anomaly log reasoning. The system uses two-tier prompting strategy which includes system prompt and user prompt.

1. System Prompt: Role Definition

The system prompt defines to role and behavior of the LLM before it process tasks. In this project, th prompt instructs the LLM to become a expert in HDFS

log analysis and anomaly detection. In addition, the prompt strictly enforce the output format requirement for LLM to JSON format only. This approach is necessary because it will prevent other element like markdown text, emojis, or code blocks that usually be generated by LLMs and keep the response in consistent format for later parsing.

2. User Prompt

The user prompt constructs a guide for LLM to analyze the log sequences that are detected as anomaly. Using the context that is prepared, the prompt let the model explore the all the related event to give a more concise explanations and explanations actions.

4.6 Frontend Implementation

This section will describes the implementation of the frontend layer of the system which provides the interface for user to upload log, visualize anomaly and analysis results. The frontend is implemented using React framework which offer the ability to build a responsive and scalable frontend for the system.

4.6.1 React Application Structure

The frontend of the system organizes the components by grouping them by functionality. This approach improves the scalability and maintainability of the code base. The primary source directory is place under `frontend/src/` and structured as follows.

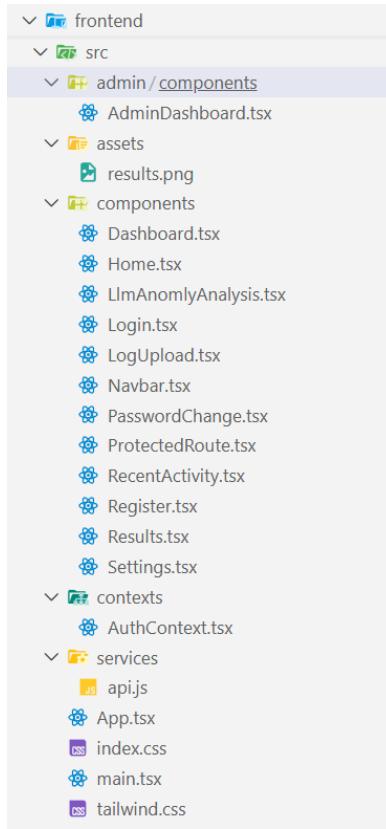


Figure 4.17: React frontend structure

Figure 4.17 shows the frontend structure of the system which provides clear separation between components to make it easy to develop and maintain. For instance, authentication-related components such as `Login.tsx`, `Register.tsx`, `AuthContext.tsx` are separated from analysis components like `LogUpload.tsx` or `Results.tsx`. In addition, the frontend is developed using TypeScript XML (`.tsx`) in stead of JavaScript XML (`.jsx`) which provides a better static type checking and support with better error detection.

4.6.2 Key Components Implementation

a. Authentication Flow

The authentication flow is responsible for verifying user identity, it also manages the session state and restrict access to routes that are protected. This will ensure only authenticated people can access to the system and perform log analysis tasks.

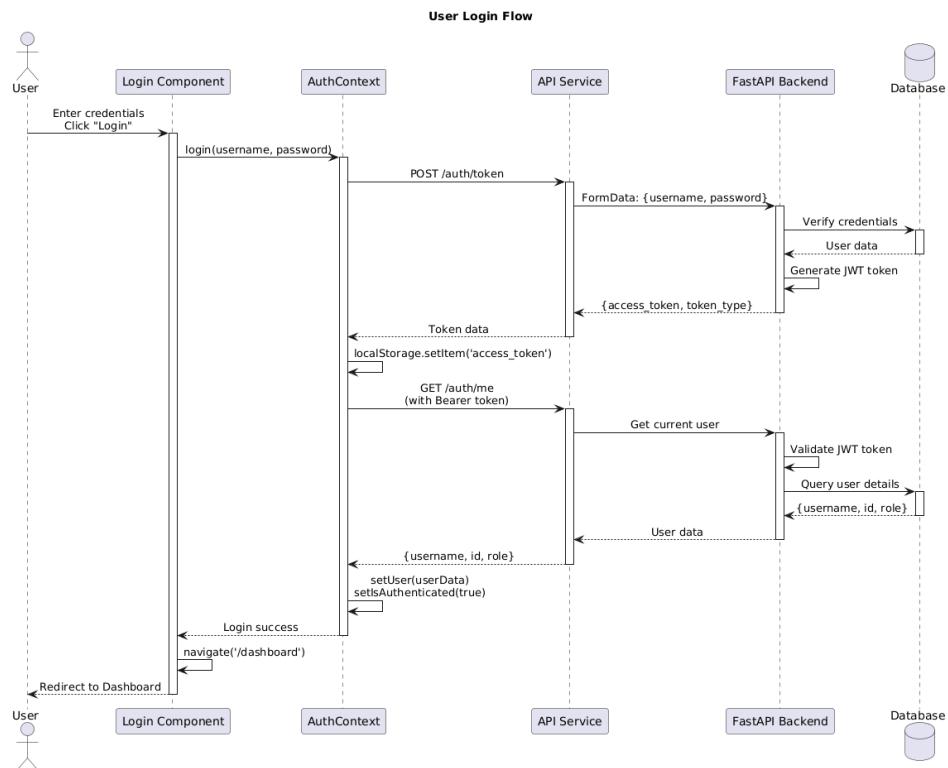


Figure 4.18: Sequence diagram for authentication flow

The Figure 4.18 describes the Authentication flow. The process begins when the user enters their credentials (username and password) into the login form of Login component and clicks "Login" button. The Login component call the login(username, password) function from AuthContext. The AuthContext then send POST request to the API endpoint /auth/token. The FastAPI backend verifies user credentials with information that stored in database. If the credentials are valid, the backend generates a JSON Web Token (JWT) and return it to the API service. The AuthContext receive the JWT, stores the token in client local storage. To retrieve the user's details, AuthContext requests /auth/me with the Bearer token and the backend validates the JWT and returns user details. Finally, The client updates authentication state and redirects the user to the dashboard.

b. Log Upload Component

The log components provides the main interface for user to submit log files for

anomaly detection. The upload interface is implemented as a React components that can handle file selection, upload progress tracking, and result viewing. The component can validate file types on the client side before send to backend to prevent unnecessary network traffic.

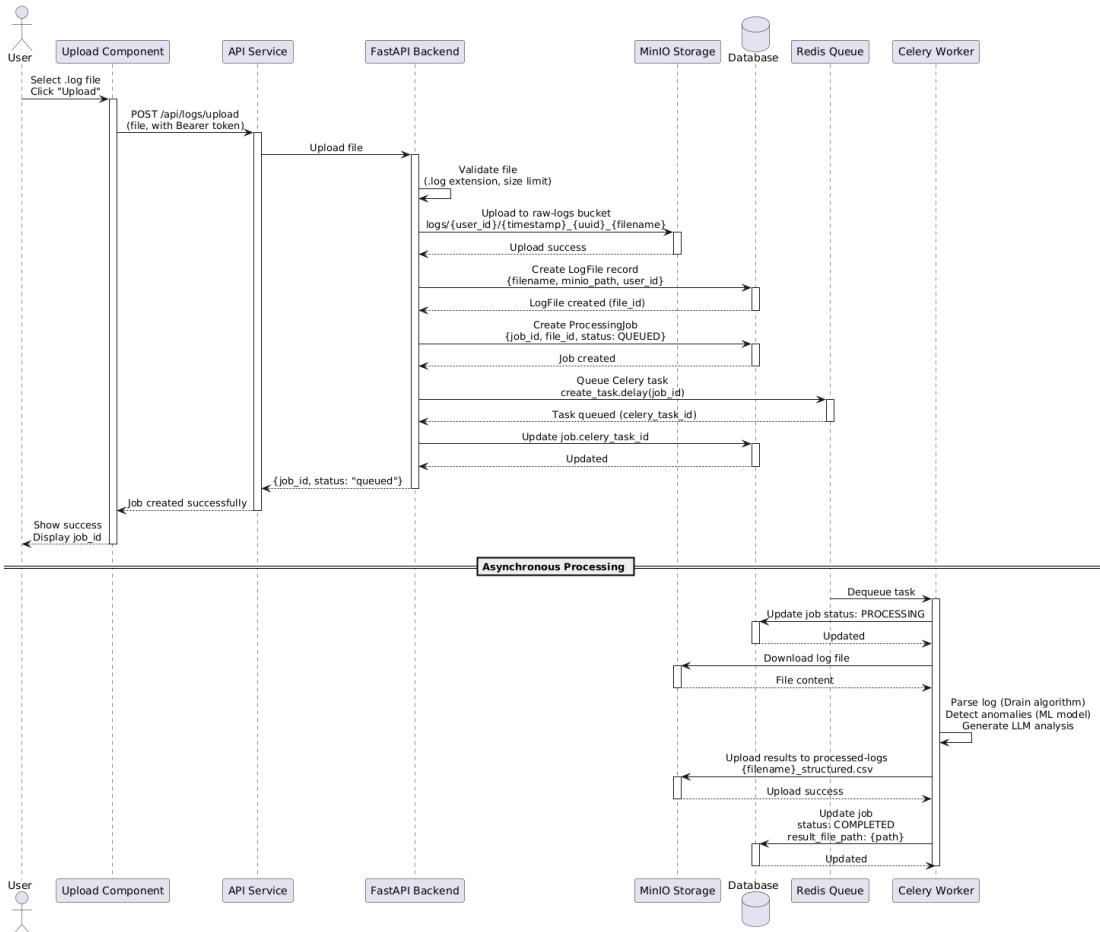


Figure 4.19: Sequence diagram for upload log

The Figure 4.19 describes the log upload phase. The process begins when the user selects a log file and clicks the upload button in the frontend Upload Component. The frontend send a POST request to the backend. The backend then valid the file then upload to MinIO storage. Once uploading success, the log processing task is created and publishing to Redis message queue. Finally, the backend returns a success response to the frontend with job ID allow user to track processing status.

CHAPTER 5: RESULT ANALYSIS

This chapter presents a comprehensive evaluation of the proposed log anomaly detection system. The analysis focuses on the overall effectiveness of the machine learning models for log anomaly detection, the overall performance of the system and the quality of LLM generated explanations.

5.1 Machine learning models evaluations

To measures the performance of each model, I use precision, recall and F-measure, which are the most commonly used metrics, to evaluate the accuracy of anomaly detection methods as we already have the ground truth (anomaly or not) for the dataset. As shown below, precision measures the proportion of how many anomalies are correctly identified in those reported.

$$\begin{aligned} \textit{Precision} &= \frac{\# \textit{Anomalydetected}}{\# \textit{Anomalyreported}} \\ \textit{Recall} &= \frac{\# \textit{Anomalydetected}}{\# \textit{All anomalies}} \\ \textit{F - measure} &= \frac{2 \times \textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}} \end{aligned}$$



Figure 5.1: Training accuracy

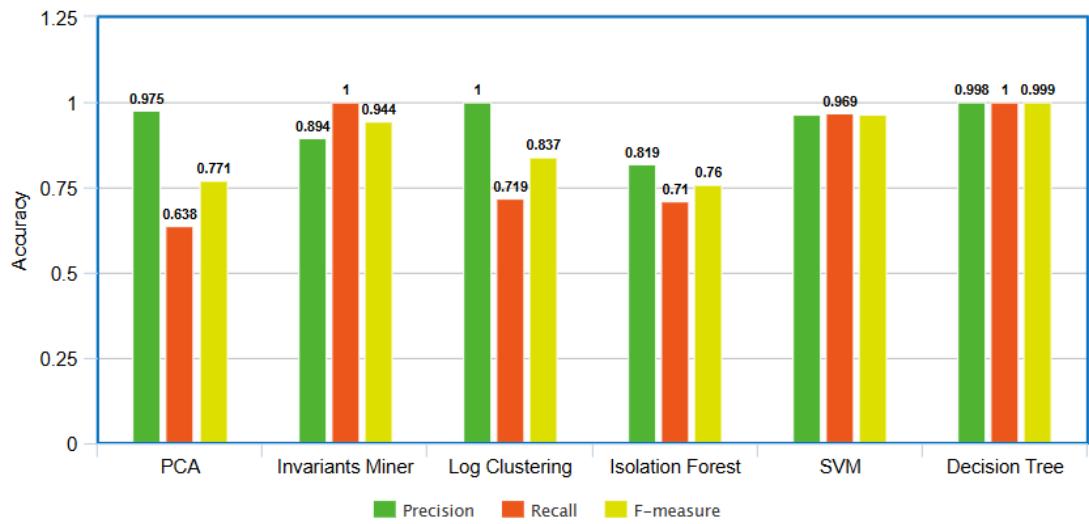


Figure 5.2: Testing accuracy

Based on the experiment results that capsules in Figure 5.1 and Figure 5.2, the Decision Tree model demonstrates that most superior performance among all approaches. It achieve near-perfect score for all three evaluated metrics on both training and testing data. SVM algorithm also perform well with F1 score approach 0.964 but the algorithm itself is more complex and harder to understand. Unsupervised methods such as PCA high precision score but low in recall, there for decrease its ability to detect diverse pattern. Forest and Logistic Regression provide moderate score but much lower than

Decision Tree model. As the result, Decision Tree model achieves the optimal accuracy and operational simplicity for my production log anomaly detection framework.

5.2 LLM responses

The LLM is provided with session-level log sequences instead of just a single log entry. This context helps improving the quality of the explanations and allows models to identify root cause more effectively. The LLM responses have a clear consistent structure which contains explanations why a log sequence is labeled anomaly, root causes analysis, severity level and recommended actions. In most cases, the recommendations are sound reliable and detail enough. Figure 5.3 show an example of generated analysis from LLM.

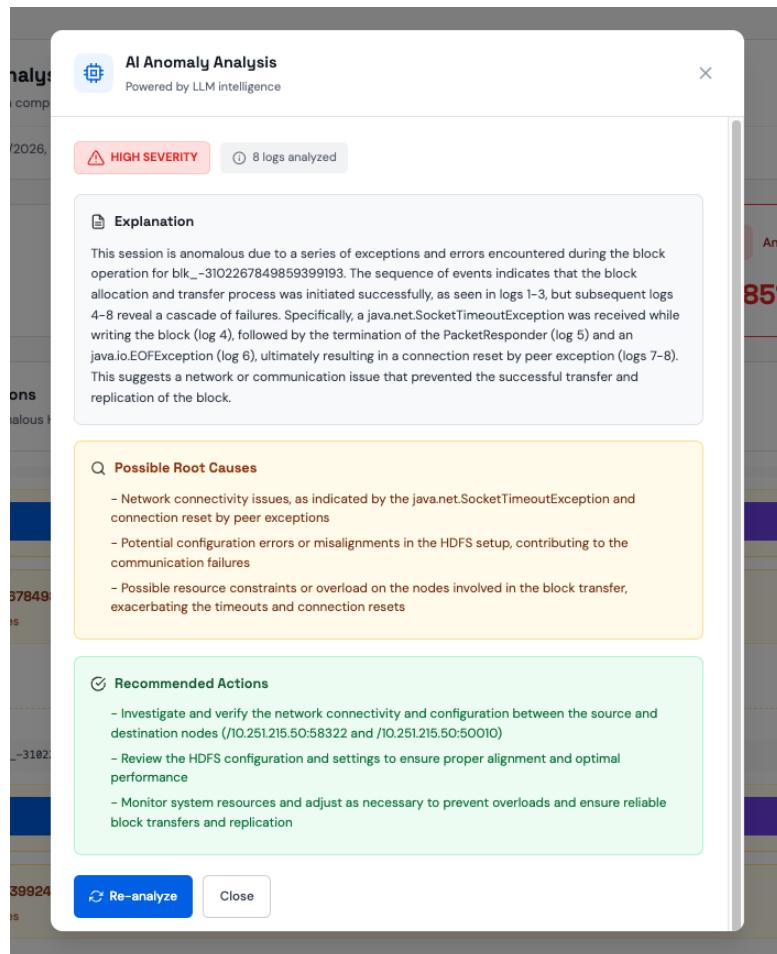


Figure 5.3: Example of a LLM response

5.3 User Interface Effectiveness

The frontend provides a clear and linear workflow: log upload, job monitoring, result visualization, and anomaly explanation review.

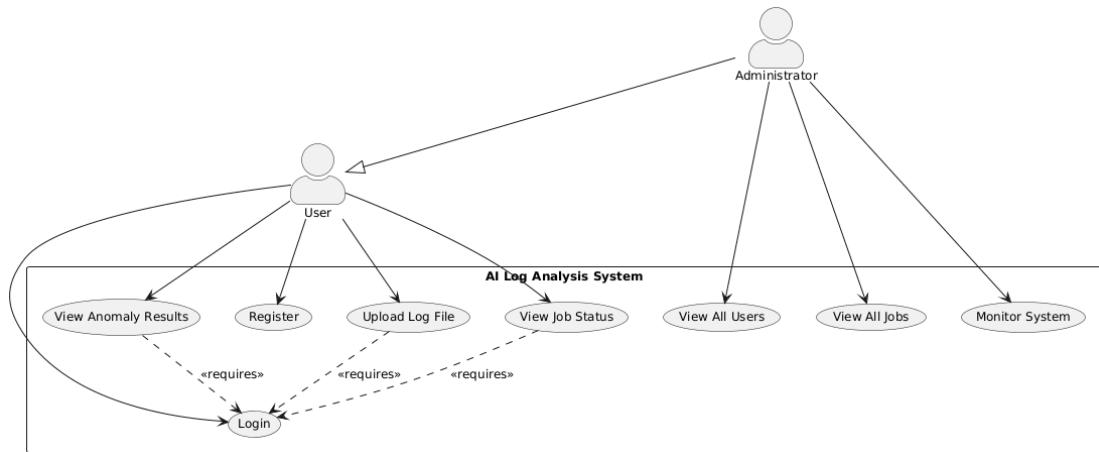


Figure 5.4: Use Case Diagram - AI Log Analysis Web Application

Figure 5.4 provides use cases that exist in this project. There are two main actors that defined in the system. Users is the primary actor who interact with the system to analyze log data and administrators who inherits all the capabilities of the standard User but also have ability to view detail of all Users and the system insights. Fundamental features that available to a user are register, login, upload log file , view job status and analyzed result.

Some main pages of the frontend which includes both User and Admin dashboard will be present as follow.

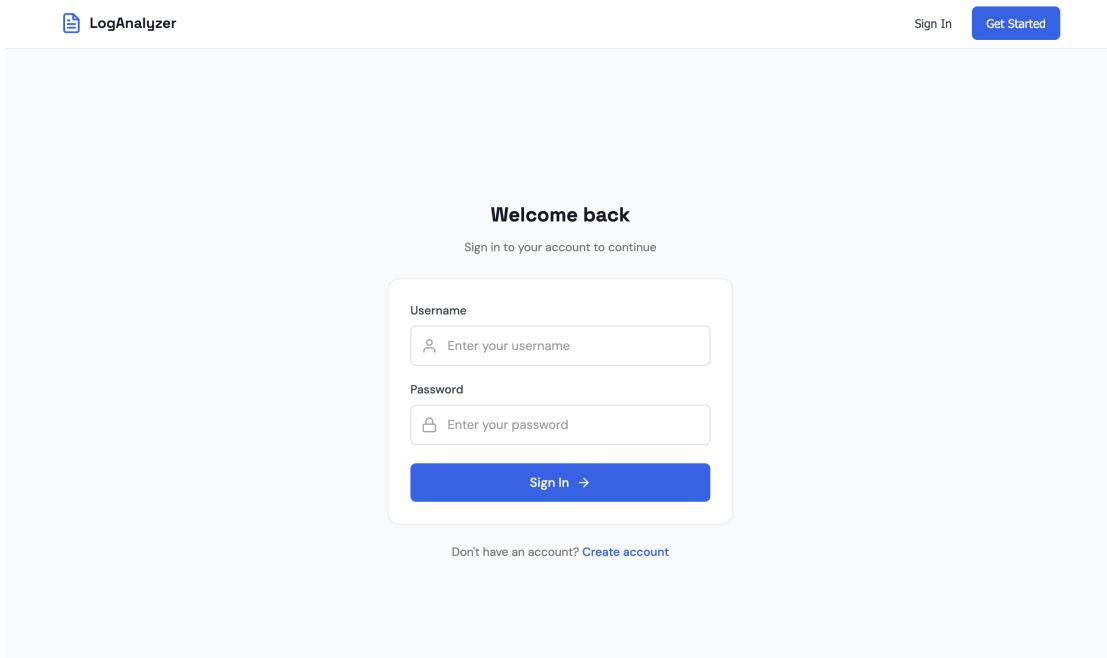


Figure 5.5: Login page

The screenshot shows the LogAnalyzer user dashboard. At the top left is the logo 'LogAnalyzer'. To the right are links for 'Dashboard', 'Profile (quan)', and 'Logout'. The main area starts with a welcome message 'Welcome back, quan' and a note 'Manage your log analysis tasks and view recent activity.' Below this are three main sections: 'Start Analysis' (with an 'Upload Logs' button), 'Recent Activity' (with a 'View Activity' button), and 'Settings' (with an 'Open Settings' button). At the bottom is a 'Getting Started' section with three numbered steps: 1. Upload Logs (description: 'Upload your log files (log, .txt) or connect to real-time streams.'), 2. Configure Analysis (description: 'Select log format and set up analysis parameters.'), and 3. View Results (description: 'Explore insights, patterns, and anomalies in your logs.').

Figure 5.6: User dashboard page

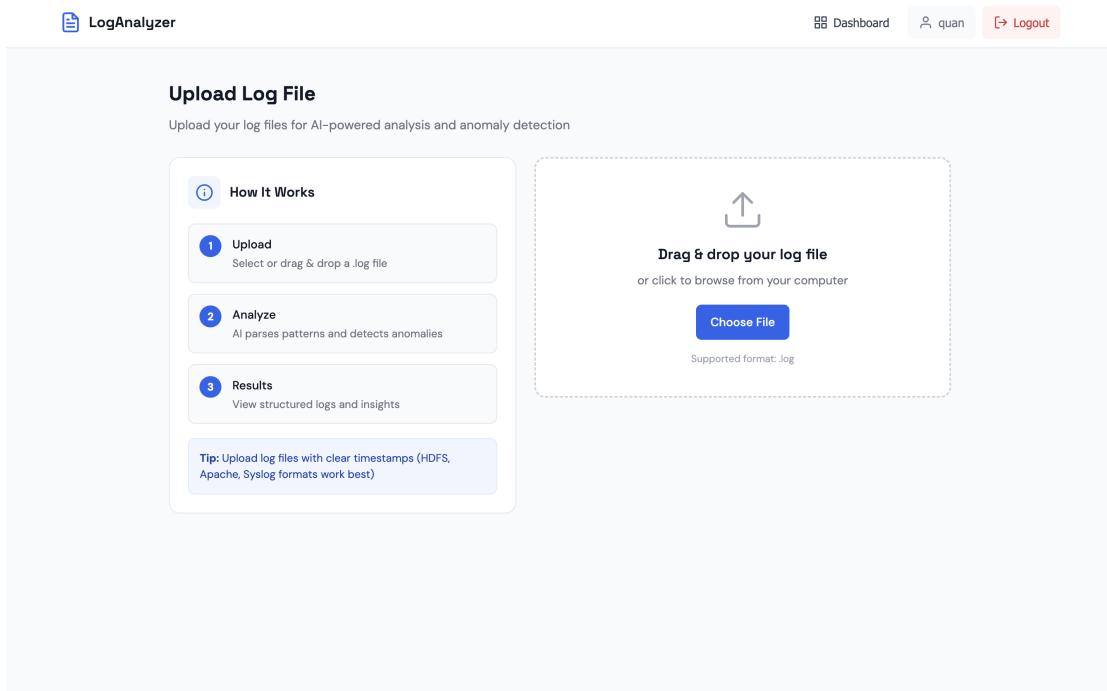


Figure 5.7: Log uploading page

Figure 5.8: LLM responses page

The screenshot shows the User Management section of the LogAnalyzer admin panel. On the left sidebar, 'User Management' is selected. The main area displays a table titled 'Registered Users' with three entries:

USER	EMAIL	ROLE
A admin ID: 2	admin@gmail.com	Administrator
M minh ID: 3	min@gmail.com	User
Q quan ID: 1	quan@gmail.com	User

Total users in the system: 3. Admin count: 1 Admin · 2 Users.

Figure 5.9: User management admin page

The screenshot shows the File Management section of the LogAnalyzer admin panel. On the left sidebar, 'File Management' is selected. The main area displays storage statistics and a chart of upload activity.

Storage Details

Total Size (Bytes)	20,145,676
Formatted Size	19.21 MB

Upload Activity
Last 30 days

A bar chart showing upload activity over the last 30 days. The Y-axis ranges from 0 to 4. The X-axis shows dates from June 1 to June 30. There are two bars: one at 1 on June 1 and one at 4 on June 29.

Figure 5.10: File management admin page

5.4 Limitations

Despite the system's promising results, there are still several limitations were identified during development and evaluation.

1. Dataset and Labeling Constraints

The system is currently relying on HDFS labeled dataset for model training, which can not accurately detect anomalies in real-world log data. If apply the models to non-HDFS logs, the result is would likely to undergo a significant degrade in accuracy. Since the supervised models can only detect patterns present in the training datam unseen or newly failures and anomalies may remain undetected unless new labeled sameples are adding the dataset for the retrain.

2. Limited Explainability

Another important limitations is the limited explainability of anomaly detection decisions produced by the machine learning models. The Decision Tree classifier determines anomaly based on the decision rules that take from event count vectors. However, these internal decision paths such as which features or rules that triggered the anomalies are not exposed to the users. As a result, the system lack of insights into each decision, making it difficult to actually validate to model behavior. While the integration of Large Language Models may mitigate the issue by providing the human-readable explainainations but yet not directly grounded in the root decision logic of the Decision Tree.

3. Manualy Interaction

The proposed system is relying on manual log upload interaction which is not practical in real-world deployment. This design limits the ability to immediately detect and response in reality where logs are generated continuously and failures must be identified as early as possible. Therefore, in practical production settings, a automated log collection solutions such as log agents or streaming pipeline are highly required to continuous ingestion and near real-time detection.

CHAPTER 6: CONCLUSION AND RECOMMENDATION

6.1 Key findings

First, the Drain parsing technique proved the effectiveness in extracting HDFS raw logs into structured data despite variations in dynamic parameters such as IP addresses and block identifiers. Second, the Decision Tree classifier showed the strong performance in detecting anomalies with near-perfect score. Third, the integration of large language models for reasoning successfully transformed classification to clear and specific explanations. However, the supervised learning approach in this study required labeled training data which may not be practical when unseen or new failures arrive.

6.2 Recommendations for future work

There are several directions for future research that are highly recommended. First, extended the framework to support multiple log formats instead of just for HDFS log in the current implementation. This would significantly improve the system's flexibility. Secondly, in the future, other machine learning approaches will be investigated to address the requirements for static, pre-labeled datasets of Decision Tree algorithms. The limitations of the current implementation could degrade the system's detection performance when deployed in real-world environments. Finally, the current implementation will be enhanced to have the ability for real-time streaming and perform more automated actions rather than relying on manual operations.

6.3 Conclusion of the research

This thesis presented a comprehensive framework for anomaly detection in HDFS logs. The research address some fundamental challenges in automated log analysis by developing a system that can combines traditional approaches with modern interpretability solution. The proposed solution implemented a end-to-end pipeline including transform raw logs into structured data using Drain algorithm, then Decision Tree classifiers is utilized to detect anomalies in log sequences and provides it to large language models for explanations and mitigation recommendations generation. This implementation can bridge the gap between anomaly detection and operational decision making. While there are still some limitations in the research such as the lack of real-time streaming capabilities, reliance on labeled training data, the study provides a practical and valuable foundation that can address challenges in automated log analysis and suggest solutions for continued development.

Bibliography

- [1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.
- [2] John Doe and Jane Smith. Anomaly detection in distributed systems. *Journal of Computing*, 10:1–15, 2023.
- [3] Elastic. Logstash grok filter documentation. <https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html>, 2024. Accessed: 2025-01-10.
- [4] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.
- [5] 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. IEEE, 2017.
- [6] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, 2016.
- [7] J Edward Jackson and Govind S Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21(3):341–349, 1979.
- [8] Ian T Jolliffe. Principal component analysis: a beginner’s guide—i. introduction and application. *Weather*, 45(10):375–382, 1990.

- [9] Richard C. T. Lee, YH Chin, and SC Chang. Application of principal component analysis to multikey searching. *IEEE Transactions on Software Engineering*, 32(3):185–193, 2006.
- [10] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):1–39, 2012.
- [11] V. Longberg. *Anomaly crowd movement detection using machine learning techniques*. Dissertation, Institution Name (if known), 2024. Dissertation.
- [12] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.
- [13] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.
- [14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.
- [15] Mohanad Sarhan, Siamak Layeghy, Nour Moustafa, Marcus Gallagher, and Marius Portmann. Feature extraction for machine learning-based intrusion detection in iot networks. *Digital Communications and Networks*, 10(1):205–216, 2024.
- [16] Splunk Inc. Using the field extractor. <https://docs.splunk.com/Documentation/Splunk/latest/Knowledge/Extractfields>, 2024. Accessed: 2025-01-10.
- [17] Hudan Studiawan, Ferdous Sohel, and Christian Payne. A survey on forensic investigation of operating system logs. *Digital Investigation*, 29:1–20, 2019.
- [18] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events

- from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794, 2011.
- [19] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126. Ieee, 2003.
- [20] Vladimir Vapnik and Vlaminir Vapnik. Statistical learning theory wiley. *New York*, 1(624):2, 1998.
- [21] 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.
- [22] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 355–366. IEEE, 2023.

Plagiarism and AI checked

The screenshot shows two side-by-side Turnitin reports. The left report is titled 'Page 2 of 77 - AI Writing Overview' and displays a 'SIMILARITY INDEX' of 12%. It includes a 'Caution: Review required.' note and a 'Disclaimer' about AI detection. The right report is titled 'thesis-ITITIU21291.pdf' and is an 'ORIGINALITY REPORT' showing a 'SIMILARITY INDEX' of 12% across various sources.

Rank	Source	Similarity (%)
1	jiemingzhu.github.io	1%
2	arxiv.org	1%
3	Zheng, Guineng. "Lifecycle of System Log Processing, Evaluation, Structurization and Application." The University of Utah Publication	1%
4	Submitted to Free University of Bolzano	1%
5	Submitted to University of South Australia	<1%
6	dspace.cvut.cz	<1%
7	www.ideals.illinois.edu	<1%
8	ir.library.osaka-u.ac.jp	<1%
9	www.eurecom.eu	<1%
10	Chunbo Liu, Mengmeng Liang, Jingwen Hou, et al.	<1%

The screenshot shows a Feedback Studio interface for a thesis titled 'Anomaly Detection in HDFS Logs using Machine Learning Integrated with LLM-Based Mitigation' by Nguyen Hoang Quan. The interface includes a sidebar with various icons for file management and a vertical toolbar on the right.

**VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**Anomaly Detection in HDFS Logs using
Machine Learning Integrated with
LLM-Based Mitigation**

By
Nguyen Hoang Quan

The thesis submitted to School of Computer Science and Engineering in partial fulfillment of the requirements of the degree of Bachelor of Engineering of Information Technology