INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Real-Time Failure Prediction in Distributed Systems via Log Analysis: A Proof of Concept

DAVIDE ANTÓNIO MELO CLEMENTE
Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Real-Time Failure Prediction in Distributed Systems via Log Analysis: A Proof of Concept

Davide António Melo Clemente

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Cybersecurity And Systems Administration

Advisor: Doutora Fátima Rodrigues

Porto, September 13, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 13, 2025

Davide António Melo Clemente

Abstract

In order to ensure high reliability in distributed systems, it is necessary to implement predictive maintenance strategies. However, manual log analysis is not a viable option when applied at scale. Modern systems and IoT devices generate a huge volume of heterogeneous logs, and unplanned outages can incur costs in the hundreds of thousands of dollars per hour. Conventional rule-based methodologies are incapable of accommodating the dynamic nature of distributed architectures; thus, there is a necessity for automated and scalable approaches. The present dissertation proposes a methodology for the transformation of raw logs into structured sequences, the extraction of informative features, and the training of machine-learning models for the purpose of predicting imminent system failures. The HDFS_v1 dataset from LogHub was utilised in the study, comprising over 11 million log lines collected from a 203-node private cloud cluster with injected anomalies. The study involved the cleaning and structuring of log sequences, as well as the derivation of unigrams, bigrams and temporal metrics. The baseline random forest and gradient-boosting models (XGBoost, LightGBM, CatBoost) are tuned via a randomised search and evaluated with 10-fold cross-validation. All models achieve macro-F1 scores above 0.98; LightGBM (0.9872) slightly exceeds XGBoost (0.9850), but statistical tests reveal no significant difference between them. The assessment of real-time performance is conducted within a simulated log-streaming environment. It is evident that both models demonstrate a high level of adaptability, with final correct-prediction ratios of approximately 0.973 (XGBoost) and 0.905 (LightGBM), and inference latencies of less than one second. LightGBM has been shown to exhibit greater flexibility, reclassifying failing blocks back to normal more frequently, whereas XGBoost has been observed to maintain failure states for a longer duration. A downtime-avoidance analysis reveals that warnings from XGBoost prevent only 3.6% of downtime (90 minutes out of 2,520 possible minutes), whereas LightGBM avoids 39.3% (990 minutes) by predicting failures earlier. The findings demonstrate the viability of machine-learning-based log analysis for predictive maintenance in distributed systems, underscore the trade-off between conservative and flexible models, and motivate future work on adaptive ensembles, confidence-based decision mechanisms and validation with production-scale data.

Keywords: Log Analysis, Failure Prevention, Machine Learning, XGBoost, LightGBM

Resumo

De modo a garantir alta confiabilidade em sistemas distribuídos, é necessário implementar estratégias de manutenção preditiva. No entanto, a análise manual de logs não é uma opção viável quando aplicada em grande escala. Os sistemas modernos bem como os dispositivos IoT geram um enorme volume de logs heterogéneos, e interrupções não planeadas podem acarretar custos de centenas de milhares de dólares por hora. As metodologias convencionais baseadas em regras são incapazes de acomodar a natureza dinâmica das arquiteturas distribuídas; portanto, há uma necessidade de abordagens automatizadas e escaláveis. A presente dissertação propõe uma metodologia para a transformação de logs brutos em sequências estruturadas, a extração de características informativas e o treino de modelos de *machine learning* com o objetivo de prever falhas iminentes no sistema. O *dataset* HDFS_v1, proveniente do projeto LogHub foi utilizado no estudo, compreendendo mais de 11 milhões de logs coletados a partir de um *private cloud cluster* contendo 203 nós com anomalias injetadas. O estudo envolveu a limpeza e estruturação de sequências de logs, bem como a derivação de unigramas, bigramas e métricas temporais. Os modelos base Random Forest e *gradient boosting* (XGBoost, LightGBM, CatBoost) são ajustados através de uma pesquisa aleatória e avaliados com validação cruzada de 10 vezes.

Todos os modelos alcançam pontuações macro-F1 acima de 0,98. O LightGBM (0,9872) excede ligeiramente o XGBoost (0,9850), contudo testes estatísticos revelam que não há diferença significativa entre eles. A avaliação do desempenho em tempo real é realizada num ambiente simulado de log-streaming. É evidente que ambos os modelos demonstram um alto nível de adaptabilidade, com taxas finais de previsões corretas de aproximadamente 0,973 (XGBoost) e 0,905 (LightGBM) e latências de inferência inferiores a um segundo. O LightGBM demonstrou maior flexibilidade, reclassificando blocos com falha de volta ao normal com mais frequência, enquanto o XGBoost manteve os estados de falha por um período mais longo. Uma análise de prevenção de tempo de inatividade revela que os alertas do XGBoost evitam apenas 3,6% do tempo de inatividade (90 minutos de 2.520 minutos possíveis), enquanto o LightGBM evita 39,3% (990 minutos) ao prever falhas mais cedo.

Os resultados demonstram a viabilidade da análise de *logs* baseada em aprendizagem automática para manutenção preditiva em sistemas distribuídos, destacam o compromisso entre modelos conservadores e flexíveis e motivam trabalhos futuros sobre *adaptive ensembles*, mecanismos de decisão baseados em confiança e validação com dados em escala de produção.

Acknowledgement

I would like to express my deep gratitude to my advisor for her guidance, patience, and dedication throughout this journey. Her support was essential to the completion of this work and to my academic growth in an area that was previously quite unfamiliar to me.

To my girlfriend, I am truly grateful for all your support, understanding and encouragement during the most difficult moments. Your presence was a constant source of motivation.

To my family, I am grateful for the trust you placed in me and for your unconditional support, which allowed me to get this far.

To my friends, I am grateful for your friendship, words of encouragement, and help whenever I needed it most.

To everyone, I offer my sincere thanks for being part of this journey and for making it possible to achieve another important milestone in my life.

"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time." - Thomas Edison

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence.
API	Application Programming Interface.
APM	Application Performance Monitoring.
BERT	Bidirectional Encoder Representations from Transformers.
BGL	BlueGene/L.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CRISP-DM	Cross-Industry Standard Process for Data Mining.
CSV	Comma Separated Values.
DBSCAN	Density-Based Spatial Clustering of Applications with Noise.
ECDF	Empirical Cumulative Distribution Function.
EFB	Exclusive Feature Bundling.
FLAP	FIU Log Analysis Platform.
GBDT	Gradient Boosting Decision Tree.
GOSS	Gradient-based One-Side Sampling.
GRU	Gated Recurrent Unit.
HDFS	Hadoop Distributed File System.
HTTP	Hypertext Transfer Protocol.
IoT	Internet of Things.
JSON	JavaScript Object Notation.
LDA	Latent Dirichlet Allocation.
LSTM	Long Short Term Memory.
ML	Machine Learning.
NLP	Natural Language Processing.
OS	Operating System.

PCA	Principal Component Analysis.
RF	Random Forest.
RNN	Recurrent Neural Network.
SaaS	Software as a Service.
SIEM	Security Information and Event Management.
SMOTE	Synthetic Minority Over-sampling Technique.
SPL	Search Processing Language.
t-SNE	T-Distributed Stochastic Neighbour Embedding.
TB	Terabyte.
XML	Extensible Markup Language.

Chapter 1

Introduction

This chapter is the introduction to the subject of this dissertation that focuses on the dissertation: predicting system failures in distributed environments through the analysis of advanced log data. While traditional methods struggle to cope with the complexity of modern systems, this analysis employs machine learning techniques to proactively identify potential issues. The chapter outlines the problem, the objectives and the methodology used to develop innovative and scalable solutions for real-time failure prediction. The aim is to enhance system reliability and minimize downtime.

1.1 Context

In recent years, we have witnessed an explosion in the growth of the data produced by increasingly complex computer systems, Internet of Things (IoT) devices, and distributed systems [1]. This expansion leads to multiple challenges related to scaling software, from data volume, a high number of requests, to thinking about how to split up functionalities in order to achieve performance, reliability, and security [2].

At the core of all these challenges, system logs present themselves as maybe the most produced type of data of all. These records serve as the backbone of observability in complex systems, capturing critical insights into application behaviour, system health, and operational anomalies. They are indispensable for debugging, performance optimization, and security monitoring, offering a granular view of interactions between distributed systems [3].

However, the sheer volume and complexity of these logs makes manual analysis impractical and reactive. Studies show that downtime in distributed systems can cost companies an average of $330,000 per hour, with large enterprises facing costs between $1 million and $5 million per hour, and mission-critical sectors sometimes exceeding $5 million per hour. These costs stem from lost revenue, productivity, recovery expenses, and reputational harm, underscoring the urgent need for automated, resilient log management and anomaly detection solutions [4, 5].

As the modern world progresses into more and more complex problems, we have come up with technologies that help us to process and analyse vast amounts of information with greater speed and accuracy than ever before. Among these technologies, Artificial Intelligence (AI) stands out as a transformative tool, capable of not only handling complex datasets, but also deriving actionable insights from them. The ability of AI to mimic human reasoning, learn from data and adapt to new challenges has made it indispensable for tackling problems in a wide array of domains, including healthcare, transportation, environmental sustainability, and beyond [6].

1.2 Problem Statement

Traditional log analysis techniques often rely on static rules or predefined patterns that struggle to adapt to the dynamic nature of modern distributed systems [7]. Furthermore, identifying meaningful patterns from massive log datasets without human intervention is a time-consuming and error-prone process. Fortunately, advanced automated techniques such as machine learning, artificial intelligence, and statistical analysis enable the analysis of large volumes of log data to detect anomalies, trends, and recurring patterns—thus reducing or eliminating the need for manual review. However, the effectiveness of these approaches depends on the quality of data, the sophistication of algorithms, and proper tuning. While they significantly improve efficiency and scalability, some level of human oversight may still be necessary to interpret complex patterns and validate results.

The growing complexity and scale of distributed systems amplify these challenges. System-generated logs are not only large in volume, but also diverse and frequently lack structure, posing challenges for traditional methods to efficiently process and analyse them. The lack of real-time insights adds to the problem, prolonging the time required for crucial interventions to prevent cascading failures and ensure system up time.

There is an urgent need for innovative and scalable solutions capable of managing the volume and variety of log data while delivering adaptive, accurate, and actionable insights in real time [8]. Although such solutions are essential to proactively prevent failures, log analysis is no easy task. Multiple researchers have already experimented with machine learning algorithms, and the results will be discussed later in the document.

1.3 Objectives

The objective of this dissertation is to develop a framework to predict system failures in distributed systems. To achieve this main goal, the following objectives were specified:

- Investigate existing techniques for failure prediction in distributed systems, including statistical methods, machine learning models, and hybrid approaches.
- Explore how generative models can be applied to system log analysis.
- Study existing tools and methods for log analysis, particularly in distributed systems. Analyse their capabilities in predictive failure modelling and limitations in real-time querying and insights.
- Design a high-level architecture for the proposed system that employs predictive failure models for real-time log analysis.
- Collect and preprocess historical logs from distributed systems. Ensure data is labelled with failure and non-failure events.
- Implement machine learning models to predict potential system failures based on historical logs and event sequences.
- Ensure that the system should provide proactive alerts when failures are predicted and allow users to explore the logs which generated the alert.
- Validate the predictive failure models using real-world data from distributed systems. Test the accuracy of predictions in both historical and real-time scenarios.

1.4 Ethical Considerations

Ethical considerations are a critical component of any research project, ensuring that the study is conducted responsibly and transparently, aligning with fundamental values of our society. The following are some aspects that rule the development of this project.

One of the key points of a project like the one proposed is keeping potential confidential data anonymized. The work should be based on a completely technical log analysis and shouldn't expose any information that could trace back to any company or service.

In light of the intricate nature of log analysis, it is imperative to emphasize that any individual utilizing the finalized iteration of the project must be conscious of the inherent constraints associated with fault prediction models. It is important to note that the model does not guarantee a 100% hit rate. Therefore, it is essential to maintain caution when monitoring, and managing services, particularly in production environments.

1.5 Methodology

In this project, we will follow the Cross-Industry Standard Process for Data Mining (CRISP-DM) methodology to analyse log data for failure prediction and system monitoring. This approach is widely used for developing projects in data mining or data processing and is designed to make the process repeatable, standardized, and effective across various industries [9–11]. During the Business Understanding phase, we will define the project's specific goals, such as identifying the main factors that cause system failures, and review relevant studies on log processing and distributed systems.

During the Data Understanding phase, we utilized an existing dataset of HDFS logs from the LogHub repository, which was originally collected by multiple researchers at the Chinese university of Hong Kong. This dataset contained log data from multiple distributed systems. We will analyse its structure and content, and assess its quality, addressing issues like missing or inconsistent entries.

The Data Preparation phase will involve cleaning the logs, selecting the most relevant data, integrating logs from different sources, and formatting the data to ensure consistency and usability for modelling. In the Modelling phase, we will then apply machine learning algorithms and refine the models through multiple experiments to optimize their performance. The Evaluation phase involves assessing the models' effectiveness in meeting project objectives and ensuring they are suitable for deployment. Finally, in the Deployment phase, we will prepare the final model for integration into operational environment, enabling real-time failure detection and system monitoring to improve response times and prevent outages.

1.6 Planning

The successful execution of this project requires a well-structured and realistic timeline. The Gantt chart below, Figure 1.1 outlines a high level overview of the key phases of the project, ensuring a logical progression and efficient allocation of time and resources.

The project begins with writing the Introduction phase (October–November 2024), during which the research problem is defined, objectives are established, and the document structure is outlined. This is followed by the State of the Art phase (November 2024–January 2025),

where existing methodologies and literature on log analysis and failure prevention using machine learning are reviewed to inform the project's approach.

Simultaneously, the Data Collection and Preparation phase (January–March 2025) focuses on acquiring, preprocessing, and exploring the data, forming the basis for model development. The Modelling and Evaluation phase (March–July 2025) involves selecting, implementing, and testing machine learning models to address the research objectives.

Throughout the project, writing the dissertation document is an ongoing activity (October 2024–September 2025), allowing continuous refinement and integration of findings into the final document. This iterative approach ensures that each phase informs the writing process and aligns with the dissertation's overall goals.

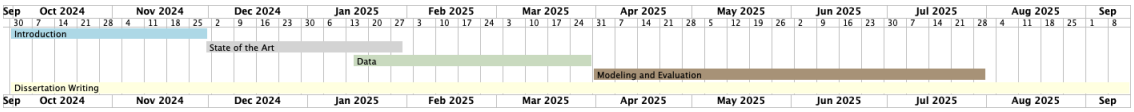


Figure 1.1: Planning Gantt Diagram

1.7 Document Structure

The structure of this dissertation has been designed to provide the reader with a logical and comprehensive progression of the research work, from the initial contextualization to the conclusions and future perspectives. The document is divided into six chapters.

- **Chapter 1: Introduction** - This chapter sets the context for the dissertation, providing a contextual framework for the subsequent discussion of the topic. The text provides an initial explanation of the problem, the objectives to be achieved, and other important considerations, including ethical aspects. The introduction also delineates the methodology and the project's planning.
- **Chapter 2: State of the Art** - This chapter is intended to provide a comprehensive overview of the existing knowledge, research, and technologies that were used in the dissertation. The document addresses fundamental concepts related to the project's objective, including logs, log analysis, distributed systems, machine learning, predictive models, evaluation metrics, and the technologies considered for project development.
- **Chapter 3: Implementation** - In this chapter, the dataset under consideration is presented, along with the procedural steps involved in its processing and the particulars of its subsequent exploration. The implementation of the proposed approach for predicting failure is described, with the structure of the implementation following the CRISP-DM methodology. This encompasses business and data understanding, data preparation, selection and training of machine learning models, hyper parameter optimization, and evaluation of their performance. Finally, it details the simulation of real-time deployment.
- **Chapter 4: Log Streaming Simulation Results** - The purpose of this chapter is to present the results obtained from applying the models to a simulated real-time deployment environment. The discussion encompasses an analysis of their strengths and an exploration of methodologies for enhancing metric values. The study encompasses the analysis of inference latency and the evolution of log block states over time.

- **Chapter 5: Conclusions** - The final chapter of the study offers conclusions regarding the work performed throughout the project. It also delineates the prospective directions and opportunities for advancement and enhancement of the research. It is suggested that improvements be made to the model and features (for example, through the exploration of deep learning architectures). Furthermore, it is proposed that the method be applied to a more extensive range of datasets and that real-world validation be conducted.

Chapter 2

State of the Art

This chapter provides exploration about core concepts related to the objective of this document, addressing content related with logs, log analysis, distributed systems, machine learning, predictive models, metrics on how to evaluate them and finally technologies that will be considered to develop the project.

2.1 Distributed Systems

As the world progressed into more and more complex computer systems, the need for addressing concerns such as modularity, separation of concerns and redundancy became increasingly important. As a result, monolithic systems, which were once the dominant paradigm, began to be replaced with distributed systems. Maarten van Steen, Andrew Tanaenbaum [12], define a distributed system as a "collection of independent computers that appears to its users as a single coherent system".

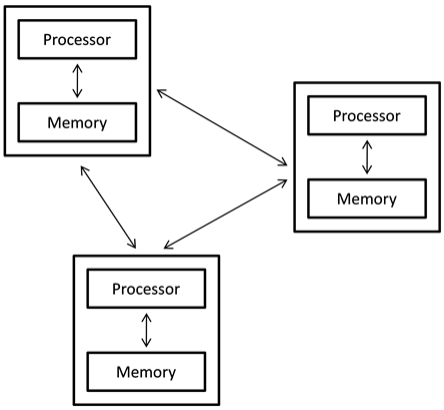


Figure 2.1: Distributed system [13]

As seen in Figure:2.1, distributed systems don't share memory, Central Processing Unit (CPU) or even Operating System (OS). In these kinds of systems, nodes exchange information over a communication medium. This information may be requested through multiple channels, including Hypertext Transfer Protocol (HTTP) requests or message queues [13].

When deciding between a monolithic or distributed system, there are several aspects to consider in favour of the latter:

- **Scalability**

  - Horizontal scaling - distributed systems can be easily horizontally up-sized by adding more nodes to the system, while monolithic services generally require more powerful hardware.
  - Geographical Scaling - distributed systems can deal with resources that are geographically far from each other, minimizing latencies.
  - Administrative Scaling - distributed services can be more easily managed given it's granularity and flexibility

- **Fault tolerance**
  - Redundancy - in a distributed systems, if one node fails, the others can continue it's normal work, thus guaranteeing the service availability. On the other hand, if a monolithic service goes down, then the whole flow can become inoperable.
  - Fault isolation - faults in a distributed system are generally (and preferably) isolated from the rest of the system, minimizing the impact as a whole.

Even though distributed systems allow for a seamless distribution of work between multiple devices, this architecture doesn't come without it's drawbacks:

- **Synchronization** - In the case that the multiple nodes need to be synchronized between them, we cannot assume that there is something like a global clock. Discrepancies in milliseconds can something make a big difference between a successful processing path or not.
- **Management** - The fact that this architecture consists of a collection of nodes, that implies that someone has to manage the membership and organization of that collection.
- **Failures** - Partial or complete fails can very easily lead to unexpected behaviour where some services can continue to execute normally while others can come to a halt.

2.1.1 The Role of Logging in Distributed Systems

Services within distributed systems often operate continuously, requiring 24/7 availability, and typically lack direct visual interaction with users to indicate their internal processes. To bridge this gap and provide insights into system operations, services generate substantial volumes of log messages. These messages are frequently accumulated in files called log files and vary significantly in format and content, reflecting the diverse nature of their originating systems. For example, logs from a printer might document operational anomalies, whereas web server logs could capture detailed traffic data, including HTTP status codes, operation outcomes, and other critical metrics [3].

Given the essential role logs play in monitoring and understanding system performance, there is significant interest among researchers, developers, and system administrators in developing robust tools for log analysis. These tools aim to structure and query log data effectively, generate metrics, and even trigger alerts to prevent potential issues. This focus highlights the evolving importance of logs in ensuring the reliability and efficiency of distributed systems.

2.1.2 Log Data Characteristics and Volume

Today, distributed systems can produce staggering quantities of log data, sometimes reaching terabytes per day. According to a recent industry report, nearly one in four organizations generate at least 1 Terabyte (TB) of log data daily, with around 12% exceeding 10 TB among large enterprises [14]. Handling such volumes requires robust ingestion pipelines and scalable storage.

Moreover, the rapid stream of log events — with metadata, traces and context - requires near real-time processing to support anomaly detection or fault recovery. Continuous ingestion at such rates puts pressure on indexing, storage and analysis subsystems.

The heterogeneous nature of log formats further complicates matters. Logs from different services, languages, and libraries often follow no fixed schema, ranging from structured JavaScript Object Notation (JSON) and key-value records to free-text messages. This heterogeneity necessitates robust parsing, normalization and filtering strategies before logs can be analysed or retained for subsequent tasks.

The sheer volume and variety of logs in the real world reinforces the need for efficient log pipelines that can scale horizontally, filter out redundancies, compress data intelligently and adapt parsers to diverse inputs, all while ensuring the timely detection of critical events.

2.2 Traditional Log Analysis Techniques

2.2.1 Rule-Based and Pattern Matching Approaches

Traditional log analysis techniques are largely based on rule-based systems and pattern-matching mechanisms. These approaches rely on manually defined rules, often implemented using regular expressions, to identify specific keywords, error codes or log patterns that signal anomalous behaviour. In operational settings, these rules are typically used to trigger alerts when certain thresholds are exceeded; for instance, when a particular error occurs more than a set number of times within a specified time frame. While these methods are straightforward and effective in controlled or well-understood environments, they struggle to cope with the complexity and variability of modern distributed systems [15]. The logs in these systems are often unstructured and highly dynamic, and are emitted from a wide range of sources with different formats and semantics. Consequently, rule-based techniques require frequent maintenance and manual updates to remain effective [15]. They are also ill-equipped to detect previously unseen anomalies or subtle correlations between events spanning components. These limitations increase the likelihood of both false positives and undetected failures, as well as imposing a significant operational overhead. Consequently, while rule-based log analysis remains useful for baseline monitoring, it is increasingly being supplemented or replaced by more flexible, adaptive, data-driven methods that can learn from historical data.

2.2.2 Log Aggregation and Centralized Logging

As systems grow in scale and complexity, it becomes essential to be able to collect, store and analyse logs effectively across multiple components. Log aggregation and centralised logging address this need by consolidating log data from distributed sources into a single location for unified processing. This simplifies log management and querying, enables holistic system visibility and supports advanced analytics, such as correlation across services. Technologies

such as Fluentd [16], Logstash [17] and OpenTelemetry collector [18] are some of the tools for implementing centralised logging architectures. They typically operate by forwarding log messages from various sources to a central server or logging platform where they can be parsed, enriched, indexed and stored.

Centralised logging offers several advantages over isolated log storage. It enables more efficient troubleshooting, allowing operators to trace issues across service boundaries and correlate events based on time or identifiers. It also supports alerting and dashboarding systems, enabling the monitoring of key metrics and the detection of anomalies in near real-time. However, this approach introduces its own challenges. Aggregating large volumes of logs can put a strain on network and storage resources, particularly when logs are emitted at a high frequency. Furthermore, the latency introduced by log forwarding and indexing can restrict the capacity for real-time analysis. Ensuring data reliability and maintaining consistent log formats across systems also requires careful configuration and standardisation. Despite these challenges, centralised logging remains a crucial component of modern observability stacks, serving as the foundation upon which many advanced monitoring and analysis techniques are built [19].

2.3 Machine Learning for Log Analysis

2.3.1 Supervised Learning for Log Analysis

Supervised learning has emerged as one of the most widely adopted approaches for log-based failure prediction, particularly in scenarios where historical data is labelled with known outcomes, such as successful operations or system failures. In this approach, a machine learning model is trained to map input features, typically derived from sequences of log events, to predefined output labels representing the system state. Once trained, the model can generalise from these patterns and make predictions on new, unseen data, enabling the early detection of anomalies or imminent failures.

The success of supervised learning in this context depends largely on the availability and quality of labelled datasets. Logs must first be transformed into a structured representation, often through a combination of log parsing, feature extraction and aggregation. Common features include the frequency of specific log events (unigrams), the co-occurrence of events (bigrams) and time-based features, such as inter-arrival intervals. Metadata, such as log level or component name, may also be included. These features are then fed into classification algorithms such as decision trees, support vector machines, random forests or gradient boosting methods like XGBoost and LightGBM. These models are well suited to high-dimensional, sparse data and have demonstrated robust performance in various real-world scenarios.

One of the key advantages of supervised approaches is their ability to learn from historical patterns and make highly accurate predictions when there is sufficient labelled data available. However, their performance can degrade significantly in the presence of class imbalance, which is common in failure prediction tasks where failure events are rare. Techniques such as resampling, class weighting or using metrics such as the F1 score instead of accuracy are often employed to mitigate this issue. Supervised models may also struggle to detect previously unseen failure modes because their predictions are constrained by the patterns observed during training. Despite these limitations, supervised learning remains a cornerstone of log analysis research and continues to underpin many industrial solutions for proactive monitoring and predictive maintenance [20–22].

2.3.2 Unsupervised Learning for Log Analysis

In many operational environments, obtaining labelled data for failure events is either impractical or impossible, making unsupervised learning an attractive alternative for log analysis. Unlike supervised methods, unsupervised techniques do not require predefined labels; instead, they aim to discover patterns, structures or deviations within the data. This is particularly valuable in anomaly detection tasks, where the objective is to identify log sequences or events that differ from normal system behaviour.

One common approach is clustering, where log sequences are grouped based on similarity and outliers — points that do not fit well into any cluster — are flagged as potential anomalies. Algorithms such as k-means, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and hierarchical clustering have been used for this purpose. Dimensionality reduction techniques such as Principal Component Analysis (PCA) and T-Distributed Stochastic Neighbour Embedding (t-SNE) are often used together to project high-dimensional log features in to lower-dimensional spaces where patterns are more easily identifiable.

Another effective unsupervised technique is the use of auto encoders, which are neural networks trained to reconstruct their input data. When applied to logs, the auto encoder learns to represent normal patterns with minimal reconstruction error. During inference, high reconstruction errors may indicate unusual or anomalous log sequences. This approach has shown promising results in detecting subtle deviations in system behaviour without the need for labelled failures.

Despite their flexibility, unsupervised methods present challenges, too. Without ground truth labels, it is difficult to evaluate their performance, which often relies on proxy metrics or expert validation. Furthermore, these techniques can be sensitive to noise and may incorrectly identify normal variations as anomalies, resulting in false positives. Therefore, careful feature selection, preprocessing and threshold calibration are crucial for effective deployment. Nevertheless, unsupervised learning remains a powerful tool for exploratory log analysis and anomaly detection in environments where labelled data is scarce or incomplete [20, 21, 23].

2.3.3 Deep Learning for Log Analysis

Deep learning techniques have gained considerable traction in log analysis due to their ability to model complex, non-linear relationships and learn hierarchical representations from raw data. Unlike traditional machine learning methods, which often rely on hand-crafted features, deep learning models can automatically extract and learn relevant patterns from sequences of log events. This makes them particularly effective at detecting anomalies and predicting system failures in large-scale environments.

Recurrent Neural Networks (RNNs) and their gated variants, such as Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, are widely used to capture temporal dependencies within log sequences. These models can learn the typical flow of log events in a system and identify deviations that signal potential anomalies.

More recently, transformer-based architectures, originally developed for natural language processing, have been adapted for log analysis. These models include attention mechanisms to capture long-range dependencies and offer significant advantages in terms of scalability and parallelisation. Techniques such as Bidirectional Encoder Representations from Transformers (BERT) have been shown to improve the capture of the semantic and structural nuances of log messages, enabling more accurate anomaly detection and sequence modelling.

2.4 Feature Engineering and Representation Learning for Logs

2.4.1 Log Parsing and Structuring

Before raw logs can be used for machine learning, they must first be converted into a structured format that accurately reflects system events. This process, known as log parsing, is essential because logs are often unstructured or semi-structured, consisting of free-text messages combined with timestamps, component names and other metadata. Parsing enables recurring patterns to be identified and structured identifiers (e.g. event IDs) to be assigned, which are critical for subsequent analysis tasks.

Historically, manual parsing based on regular expressions or domain-specific rules was employed, but these methods are error-prone and challenging to maintain in dynamic environments where log formats frequently evolve. More recent approaches address these limitations by relying on automated parsing techniques that treat the problem as a form of structural clustering. One of the most widely recognised tools in this area is the Log Parser project [24] from LogPAI, which provides a unified benchmarking platform and includes parsers such as Drain [25], IPLoM [26] and LogCluster [27]. These tools automatically extract log templates and assign event IDs to raw messages, significantly reducing noise and redundancy.

While this dissertation uses preprocessed logs from the LogHub [28] repository, evaluating tools such as Log Parser was instrumental in understanding the current state of log structuring. Such tools play a foundational role in real-world pipelines, where raw logs must be parsed in real time to support anomaly detection, root cause analysis and predictive modelling.

2.4.2 Feature Extraction Techniques

Once the logs have been structured — usually into sequences of event identifiers — the next step in the analysis process is to extract features. This process transforms the sequences into numerical representations that can be used as input for machine learning models. Well-designed features are essential for optimal model performance as they capture meaningful patterns that distinguish between normal and anomalous system behaviour.

A common approach is to extract event frequency features, counting the number of times each log event appears in a sequence. These frequency-based features, often referred to as 'uni-grams', capture the presence and recurrence of specific system events. To complement this, sequential features such as bigrams or trigrams can be included to preserve order information by recording pairs or triplets of co-occurring events. This is particularly useful in distributed systems, where certain event transitions may indicate underlying issues [29, 30].

Another important category is temporal features, which leverage the timing between consecutive events. Metrics such as the total duration of a sequence, the average time between events or the maximum time gap can reveal abnormal delays or slowdowns in the system. These features are particularly useful for identifying performance anomalies and early signs of failure.

Alongside these core strategies, some studies incorporate features based on log levels (e.g. INFO, WARN, ERROR), component identifiers or position-based encodings. However, the choice of features depends heavily on the structure of the logs, the prediction objective and the availability of labelled data.

Overall, effective feature extraction transforms raw sequences into rich representations that enable learning algorithms to detect subtle behavioural patterns, making this one of the most impactful steps in any log-based prediction pipeline.

2.4.3 Representation Learning for Logs

Representation learning has become an increasingly important technique in log analysis, particularly in scenarios where manually engineered features are insufficient to capture the complex semantics and relationships within the data. Unlike traditional feature extraction methods, which rely on predefined rules or statistical summaries, representation learning aims to automatically learn dense, low-dimensional embeddings of log events or sequences that preserve meaningful structure and context.

Initial aproaches in this area adapted techniques from natural language processing, such as *word2vec* and *doc2vec*, to create vector representations of individual log events or entire sessions. These embeddings capture syntactic and semantic similarities, enabling similar log patterns to be positioned closer together in the learned feature space. This approach has proven useful for downstream tasks such as clustering, anomaly detection and failure prediction [31].

More recent work has introduced specialised techniques for logs, such as *template2vec* and *log2vec*, which learn embeddings from parsed log templates rather than raw text. These methods consider the structure of the tokens and the underlying semantics of log messages, often incorporating log metadata (e.g. component type or log level) into the learned representation.

Representation learning is also commonly integrated with deep learning architectures, such as LSTMs or Transformers. These models can learn contextualised representations by encoding the temporal dependencies and sequence structures of logs, enabling more accurate modelling of system behaviour. Approaches involving pre training, where embeddings are learnt on large unlabelled datasets and then fine-tuned on task-specific data, can further improve the flexibility and robustness of the learnt features [32].

Overall, representation learning allows for a richer and more flexible modelling of log data, reducing reliance on manual preprocessing or domain-specific feature engineering. As log volumes and complexities continue to increase, these techniques play a pivotal role in scaling log analysis systems while ensuring high levels of accuracy and adaptability.

2.5 Evaluation and Validation Techniques

2.5.1 Metrics for Log Prediction

Evaluating the effectiveness of log prediction models, particularly those used for failure prevention or anomaly detection, requires metrics that go beyond simple accuracy. This is especially true given the significant class imbalance that characterises most operational datasets. In many real-world settings, normal behaviour dominates log sequences, while failures or anomalies represent a small minority. Consequently, models can achieve deceptively high accuracy by predicting only the majority class.

To overcome this limitation, evaluation typically focuses on precision, recall and the F1 score. Precision quantifies the proportion of correctly predicted failures or anomalies, while

recall measures the proportion of true incidents that were successfully identified. The F1-score, the harmonic mean of precision and recall, provides a balanced indicator of predictive performance, especially when both false positives and false negatives are costly.

In highly imbalanced scenarios such as failure prediction, the macro-averaged F1-score is widely used. This metric computes the F1-score independently for each class and then averages the results, treating each class equally regardless of its frequency. This avoids the bias toward the majority class seen in micro-averaged metrics or simple accuracy measures. This approach provides a more balanced assessment of model performance across both the majority and minority classes, ensuring that the model effectively identifies failure events while minimizing false alarms.

2.5.2 Statistical Significance Testing

When it comes to log prediction, differences in model performance can seem significant when comparing metrics such as the F1 score or recall. However, it is important to critically examine these differences to determine whether they reflect genuine improvements or are merely the result of variability introduced by the data or model initialisation. Statistical significance testing provides a systematic approach to evaluating the reliability of such performance improvements.

When models are evaluated using repeated cross-validation or multiple random seeds, a distribution of scores is obtained rather than a single deterministic result. Statistical tests use these distributions to determine whether one model consistently outperforms another. For instance, the paired t-test compares mean performance across multiple runs under the assumption that differences are normally distributed. If this assumption is not met, the Wilcoxon signed-rank test is a robust non-parametric alternative that only considers the ordering of differences, not their magnitude.

To assess whether there was a statistically significant difference in performance between the models, a two-step hypothesis testing procedure is conducted, following established statistical validation practices.

1. **Normality Test — Shapiro-Wilk**

   The first step involved verifying whether the fold-wise differences in F1-macro scores between the two models follow a normal distribution. For this purpose, the Shapiro-Wilk test is applied, with the following hypotheses:

   - H_0: The differences are normally distributed.
   - H_1: The differences are not normally distributed.

   The test outcome is determined based on the p-value. If $p > \alpha$ (with a significance level of $\alpha = 0.05$), the null hypothesis is not rejected, and normality is assumed. Otherwise, the null hypothesis is rejected, indicating that the data do not follow a normal distribution.

2. **Statistical Test for Paired Samples**

   Depending on the outcome of the normality test, one of the following tests is applied to compare the performance of the models:

   - If the differences are normally distributed, a paired t-test is performed

- If the normality assumption is violated, the Wilcoxon signed-rank test is used instead

The hypotheses tested are:

- H_0: There is no significant difference between the models.
- K_1: There is a significant difference between the models.

If the resulting p-value exceeds the chosen significance level ($p > 0.05$), H_0 is not rejected, suggesting that there is no statistically significant difference in performance between the two models. Conversely, if $p \leq 0.05$, H_0 is rejected, indicating that the observed performance difference is statistically significant.

2.6 Existing Tools and Frameworks

2.6.1 Commercial and Open-Source Log Management Platforms

Log management platforms offer a centralised system for collecting, indexing, searching and analysing machine-generated data. Effective log management is crucial for debugging, performance monitoring and security, given that modern infrastructures generate substantial volumes of time-stamped events. Commercial platforms often combine log management with metrics and tracing to offer a unified observability and security solution. This section focuses on Datadog [33] and Splunk [34], two market leaders with similar objectives, but with very different architectural and pricing approaches.

2.6.2 Datadog

Datadog is a cloud-native Software as a Service (SaaS) platform designed to unify infrastructure monitoring, Application Performance Monitoring (APM) and log management. The Log Explorer feature offers full-text search, live tail for streaming logs, analytics (such as aggregations and filtering), and pattern detection [35]. Datadog integrates with a wide range of technologies. Its documentation states that there are around 850 built-in integrations across cloud services, containers, and databases [36]. Data is collected via lightweight agents installed on hosts or containers, as well as through ingestion Application Programming Interfaces (APIs). Pre-built dashboards provide immediate insight into performance metrics and logs, and the Watchdog feature uses machine learning to automatically detect anomalies and generate alerts. Datadog also offers a Cloud Security Information and Event Management (SIEM) module that correlates security events with logs and metrics [37].

Datadog's strengths include ease of deployment (as it requires no on-premises infrastructure), a rapid time-to-value, and a rich integration library. These qualities make it appealing to DevOps teams and organisations that carry out most of their work in the cloud. However, Datadog's SaaS model may not meet strict data residency or compliance requirements, and its usage-based pricing (per host and per gigabyte ingested) can become expensive at scale. Additionally, the platform's search and query capabilities are considered less flexible than Splunk's Search Processing Language (SPL) [38].

2.6.3 Splunk

Splunk started out as an on-premises log search tool and has evolved into a unified data platform, which is offered as a self-hosted solution (Splunk Enterprise) or as a cloud-based

service (Splunk Cloud). Its distributed architecture consists of universal forwarders that collect and forward data from sources such as syslog, Comma Separated Values (CSV) and JSON files, indexers that parse and store the data for efficient retrieval, and search heads that provide query processing and visualisation [39].

Splunk's proprietary SPL enables users to create complex queries involving filtering, transformation, statistical analysis and field extraction. The platform includes customisable interactive dashboards via Extensible Markup Language (XML) or a web UI, a marketplace offering hundreds of add-ons for third-party systems, and optional machine learning toolkits. Splunk can handle both structured and unstructured data through schema-on-read indexing, making it suitable for complex environments.

Splunk's strengths include its flexible deployment options (on-premises, in the cloud, or in a hybrid environment) and the analytical power of SPL, which supports in-depth correlation analysis and the creation of custom metrics. Its architecture can handle large volumes of data and offers robust integration and security features. However, installing and managing Splunk is more complex than using SaaS tools, as administrators must configure forwarders, indexers and search heads. The pricing model, which is based on indexed data volume and optional modules, can be expensive and complicated. Furthermore, new users may need to build dashboards and alerts from scratch and face a learning curve with SPL [37].

2.6.4 Comparison

The table below summarises Datadog and Splunk across multiple dimensions.

Table 2.1: Comparison between Datadog and Splunk

Feature	**Datadog**	**Splunk**
Deployment model	SaaS only	On-premise, cloud, hybrid
Data ingestion	Agent-based + APIs (850+ built-in integrations)	Forwarders + Splunkbase add-ons
Querying	Basic filtering and analytics	Advanced queries with SPL
Setup & onboarding	Fast (no infrastructure to install)	Complex (requires forwarders, indexers, search heads)
Dashboards & UI	Pre-built dashboards	Highly customisable (requires setup)
Machine learning	Built-in (Watchdog for anomaly detection)	ML Toolkit + custom model support
Security & SIEM	Cloud SIEM module available	Mature SIEM capabilities with fine-grained control
Pricing	Per host + per GB ingested (usage-based)	Based on indexed data volume + modules
Best for	DevOps teams, cloud-native environments	Enterprises with compliance/security needs
Limitations	SaaS only, costly at scale, limited query power	Steep learning curve, complex pricing/setup

Datadog offers a compelling solution for organisations seeking a managed, cloud-native observability stack with minimal setup and extensive integrations. However, those requiring flexible deployment options, powerful search capabilities and advanced analytics may prefer

Splunk, despite its greater complexity and cost. The choice between the two depends on factors such as infrastructure footprint, budget, compliance requirements, and the need for custom analytics.

2.6.5 Research-Focused Log Analysis

In the domain of log analysis and failure prevention, Machine Learning (ML) has emerged as a transformative approach. This section synthesizes existing literature to provide a comprehensive overview of methodologies, tools, and advancements in the field. Log analysis has gained significant attention in research and industry due to its potential for diagnosing system behavior, identifying anomalies, and predicting failures. Existing literature highlights the different approaches adopted, ranging from traditional rule-based systems to more advanced ML methods. The integration of Natural Language Processing (NLP) techniques and deep learning has also shown promising results in interpreting unstructured log data effectively.

However, several challenges persist. Logs are often extensive, heterogeneous, and noisy, making preprocessing a critical yet difficult step. The dynamic nature of log formats across systems further complicates this processing. Additionally, the discrepancy between normal and abnormal logs poses difficulties for supervised ML approaches, often needing semi-supervised or unsupervised techniques.

Despite these difficulties, efforts have demonstrated substantial improvements in fault detection, performance monitoring, and predictive maintenance. Recent research also emphasizes interpretability and real-time analysis to enhance system reliability.

For example, the article in [40] proposes an end-to-end framework called LogM for failure prediction and diagnosis of Hadoop platforms through log analysis. LogM is composed of two modules:

- **LogM for failure prediction:** This module utilizes the CAB net which is a deep learning model that leverages a Convolutional Neural Network (CNN) with an attention-based Bi-LSTM. This model is able to predict system failures based on sequential log data and can report this information to operators to prevent issues.
- **LogM for failure diagnosis:** This module aims to find the root cause of the failure through the analysis of multiple components logs. To do so, the module performs correlation analysis, which groups logs that have common identifiers and variables from the same component within a given time interval, and leverages an anomaly knowledge graph to assist in the diagnosis.

The researchers used a real-world dataset of logs from a commercial Hadoop platform to evaluate LogM. This dataset was composed of over 50 million logs from six components and was collected over a month. The results show that LogM is highly effective at predicting and diagnosing Hadoop platform failures. The CAB net outperformed other deep learning models in failure prediction. While the supervised failure diagnosis approach performed slightly better than the unsupervised approach, both approaches achieved high accuracy.

OptimizeLog, proposed in [41] is a another model for anomaly detection and localization in distributed systems. The model uses a dictionary and length-based log parser that is more stable than other log parsers and does not require parameter tuning. OptimizeLog uses an ALBERT-based semantic embedding, count embedding, and time embedding, combined with an attention-based Bi-GRU model to improve anomaly detection accuracy. Finally, OptimizeLog uses a forest of component instances and a tree-based depth-first traversal

algorithm to locate anomalies. Experimental results show that OptimizeLog's log parsing method is 4% more accurate than other log parsers. Compared to other advanced methods such as DeepLog, LogAnomaly, and LogRobust, OptimizeLog improves the accuracy of anomaly detection by 5%, while enabling instance-level anomaly localization on real datasets.

The article by [42] proposes a new method for detecting anomalies in system logs of power microservices. The method is based on a semi-supervised approach that uses a combination of Bi-LSTM and BERT with an attention mechanism.

The authors argue that traditional methods for log anomaly detection often rely on manual rules or supervised learning techniques that require large amounts of labeled data. These methods can be time-consuming and ineffective, especially in the context of microservices, where the logs can be complex and constantly evolving.

The proposed method addresses these challenges by using a semi-supervised approach that only requires a small amount of labeled data for training. The method works by first parsing the raw log data to extract log keys or log templates. Then, BERT is used to extract semantic information from the log keys, which is then fed into the Bi-LSTM network. The Bi-LSTM network is trained to predict the next log key in the sequence. If the predicted log key is different from the actual log key then it is considered an anomaly. The attention mechanism is used to improve the performance of the model by focusing on the most important parts of the log sequence.

The authors also evaluated their method on a dataset of Hadoop logs and compared it to several other log anomaly detection methods. The results show that the proposed method outperforms the other methods in terms of recall and F-measure, while achieving similar accuracy to PCA and DeepLog. The authors conclude that the proposed method is an effective and efficient approach for detecting anomalies in power microservices logs.

This paper in [43] investigates the impact of NLP on log parsing for anomaly prediction. The authors used two datasets: an industrial aeronautical system log and a public benchmark from an Hadoop Distributed File System (HDFS) cluster.

The authors explored the following log parsing techniques:

- **Tokenization:** Breaking down text into individual words for comparison.
- **Semantic techniques:** This included synonym replacement, stemming, and stop words removal to refine the input text and reduce noise by considering word meanings.
- **Vectorization:** Transforming sentences into a vectorized representation using bag-of-words, bi-gram, and tri-gram models.
- **Model compression:** Using a hashing trick method to reduce model size and enable faster computations.
- **Classification:** Categorizing log messages using bisecting k-means and Latent Dirichlet Allocation (LDA).

For log mining, the study focused on a specific failure in the industrial dataset. The log messages were treated as a time series, and the features associated with each message were based on the chosen classification method. A Random Forest (RF) algorithm was used for classification due to its effectiveness in handling class imbalance and noise.

The paper had some key findings such as:

- High precision in failure prediction was observed across various log parsing techniques, with some combinations achieving near-perfect precision.
- Recall rates varied significantly depending on the chosen techniques. The best recall was achieved with LDA and bi-gram.
- The best overall performance (F-score) was achieved by combining LDA, bi-gram, stemming, and synonym replacement. This combination demonstrated robustness and effectiveness.

When applied to the public HDFS dataset, the best-performing combination significantly improved the F-score compared to existing methods. The authors suggest that NLP-based methods introduce flexibility, resulting in a similar human interpretation of log messages and thus enhanced performance.

In conclusion, the study highlights the positive impact of incorporating NLP techniques in log parsing for anomaly prediction. The authors recommend further exploration of advanced NLP methods and emphasize the need for more comprehensive benchmark datasets for evaluation.

Another tool, FIU Log Analysis Platform (FLAP) [44] is an integrated system for analyzing event logs. The authors argue that existing log management products lack either the generality to handle various types of event logs or the comprehensiveness to provide advanced data mining solutions. FLAP addresses these challenges by supporting the entire process of event analysis, from pre-processing to visualization, and by incorporating a number of advanced data mining techniques.

FLAP's main components include:

- **Event Pre-processing Layer:** This layer extracts events from unstructured or semi-structured raw logs using either rule-based event extraction or unsupervised template learning (clustering based on log format and structural information).
- **Event Storage Layer:** This layer stores and retrieves the extracted events in a specified repository.
- **Event Mining Libraries:** It is the core of the system. It integrates various event processing and mining algorithms such as multi-resolution event exploration, temporal dependency mining, event temporal lag mining, event summarization, and system failure prediction.
- **Event Visualization Portal:** Provides user-friendly interfaces to present analysis results intuitively, including a dashboard for basic statistics and a Dynamic Query Form for interactive data exploration.

The paper evaluates FLAP through a case study using event logs from an internal network of Huawei Technologies Co. Ltd.. The findings demonstrate FLAP's effectiveness in:

- **Event Summarization:** Providing a global overview of the system's behavior and highlighting frequent events.
- **Root Cause Mining:** Identifying potential causes of critical events using temporal dependency pattern mining, enabling effective troubleshooting.
- **Failure Prediction:** Predicting system failures with a 75% success rate, leading to reduced downtime.

The paper in [45] introduces LogAnomaly: a new framework for detecting anomalies in log data. The authors observed that:

- Log templates, predefined by developers, typically characterize the event that occurs in the system.
- Program execution flows in a service system usually have some patterns.

Based on these observations, the authors developed LogAnomaly to learn both sequential and quantitative patterns of the logs offline, and detect anomalies in real time.

This tool uses a method called **template2Vec** to extract the semantic and syntax information from log templates. This method converts the words inside templates, into word embedding vectors and calculates template vectors by combining word vectors. This allows the model to capture both word context and semantic information, including synonyms and antonyms. The authors claim that this is the first anomaly detection framework considering semantic information of logs.

The framework also addresses the challenge of new log templates appearing between training periods by matching new templates to existing ones based on the similarity of template vectors.

LogAnomaly was evaluated on two benchmark datasets: the HDFS dataset and the BlueGene/L (BGL) dataset. The results show that LogAnomaly outperforms existing log-based anomaly detection methods. It achieved the best accuracy among the five methods tested, having an averaged F1 score of 0.96 on the BGL dataset, and 0.95 on the HDFS dataset. The authors also demonstrated the benefits of combining sequential and quantitative patterns for anomaly detection.

Further evaluation was conducted on the performance of LogAnomaly in addressing new types of logs at runtime. The results show that LogAnomaly is more accurate in online anomaly detection than the baseline method Deeplog. A case study on a real-world anomaly in an aggregation switch further validated the effectiveness of LogAnomaly in detecting anomalies and reducing false alarms. The paper concludes that LogAnomaly is a promising system for log anomaly detection that can effectively extract semantic information from log templates and handle new types of logs.

This chapter has provided a comprehensive overview of the current state of log analysis for failure prediction, emphasising the strengths and limitations of various techniques. While conventional methodologies encounter challenges in dealing with the intricacy of modern distributed systems, machine learning has emerged as a promising approach. It is evident that techniques such as feature engineering, data balancing, and model selection have proven to be of critical importance in enhancing the accuracy of detection. The potential of deep learning techniques to capture the complex nuances of system behaviour is well-documented; however, their complexity requires significantly more computational resources for training and inference.

The present study will focus on leveraging supervised learning techniques, using random forest and gradient boosting algorithms, to build upon this foundation, with the goal of combining the benefits of the established performance of such models with the agility necessary to perform in real time and in an operational environment. By focusing on real-time

performance, a gap that is generally overlooked by the literature, this work will aim to improve the existing approaches and evaluate not only the detection accuracy but also the performance and agility of the solution for detecting system faults.

Chapter 3

Implementation

This chapter outlines the implementation of the proposed failure prediction approach, which is structured according to the CRISP-DM methodology. Following this methodology ensures a clear and rigorous description of the steps taken to develop the solution, including understanding the problem domain and data, preparing the dataset, selecting and training machine learning models, evaluating their performance and deploying the chosen model in a simulated real-time environment.

3.1 Business Understanding

Modern distributed systems generate vast amounts of log data that are crucial for monitoring and debugging purposes, as well as for ensuring system reliability. However, the sheer volume and complexity of these logs makes manual analysis impractical and reactive. Since failures in distributed systems can result in costly downtime and degraded performance, there is clearly a need for automated approaches that can proactively identify failure patterns.

This project addresses the problem of predicting system failures through log analysis. The central goal is to investigate how machine learning can be applied to historical logs to anticipate failures before they occur, enabling preventive action to be taken. The focus is on analysing structured logs from distributed systems and developing a pipeline that can ingest logs, learn from past failure events and provide real-time classification during system execution.

The scope of this work includes:

- preprocessing logs to extract meaningful representations of system behaviour;
- experimentation with supervised learning models for binary classification (failure vs. normal);
- validating models using rigorous metrics and statistical testing;
- simulating a real-time environment in which logs are streamed and evaluated instantly.

3.2 Data Understanding

The effectiveness of machine learning models in log-based failure prediction heavily depends on the quality, representativeness, and structure of the data used for training and evaluation. In this project, an open-source log dataset was selected to simulate a realistic environment while enabling reproducibility and benchmarking.

The dataset was obtained from the LogHub [28] repository, which is widely recognized in the research community for providing standardized system log datasets derived from real-world large-scale systems.

The present repository is a product of research in log analysis, originating from two particular papers:

- Detecting Large-Scale System Problems by Mining Console Logs [46].
- Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics [47].

LogHub offers preprocessed and raw logs from various software systems, including distributed file systems, operating systems, container orchestration platforms, and messaging services. These datasets are commonly used for research in log parsing, anomaly detection, failure prediction, and root cause analysis.

For this work, the logs generated by the Hadoop Distributed File System (HDFS) were selected due to their relevance, availability of labelled failure events, and the presence of complex system behaviours typical of distributed environments.

3.2.1 HDFS Logs

The HDFS dataset in LogHub consists of logs collected from a private cloud environment using benchmark workloads. The cluster experienced both normal and faulty behaviours, manually injected by the users (machine down, network disconnection and disk full), which were captured in the logs. This makes the dataset particularly suitable for training and evaluating failure prediction models. In the HDFS files, there are three datasets available to download: HDFS_v1, HDFS_v2 and HDFS_v3. In this project we are using the HDFS_v1 dataset.

The key characteristics of the HDFS_v1 dataset include:

- **Log volume:** Generated on a 203 node HDFS, with over 11 million log lines, span over 38h.
- **Structure:** Each log line includes a timestamp, log level (e.g., INFO, WARN, ERROR), component/module name, a message body and finally the block_id to which it belongs. Example:

  ```
  081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block
  blk_-1608999687919862906 src: /10.250.10.6:40524
  dest: /10.250.10.6:50010
  ```

- **Event labelling:** The dataset contains manually labelled sequences that are classified as either normal or anomalous. Anomalies typically correspond to system failures or behaviours that precede them.
- **Auxiliary files:** The download of the dataset also includes extra files that were already pre-processed:
  - `anomaly_label.csv`: Mapping of each block_id to the final execution result (Normal or Anomaly);
  - `event_occurrence_matrix.csv`: The amount of occurrences of each event type within each block, as well as the end result.

- `log_templates.csv`: The different event ids, mapped to the correspondent message template. E.g. `E1`,[*]Adding an already existing block[*]
- `event_traces.csv`: This was the mainly used file. It contained the whole sequence of events within each block, the latencies between events, as well as the end result. This file alone allowed to extract most of the relevant features such as unigrams, bigrams and temporal metrics, as explained in greater detail in 3.3.2.

3.3 Data Preparation

Prior to the training of any model, a series of preprocessing steps were implemented to take the log dataset into a structured format that was conducive to machine learning. The steps involved in this process included data preparing and cleaning, feature extraction, class imbalance and label codification.

3.3.1 Data preparing and cleaning

All log entries were first checked for consistency, ensuring that timestamps followed a valid format and that no corrupted or incomplete entries were present. Duplicate log lines, which could bias event frequency statistics, were removed. Only entries containing a valid mapping to an event identifier were retained, ensuring a consistent representation of system behavior. The main file used was `Event_traces.csv` that contained columns such as

- **BlockId** - Log block identifier
- **Events** - Sequence of `event_id`
- **Label** - Success or Fail
- **TimeInterval** - List of time intervals between events

Table B.1 depicts a a sample of data contained in the `Event_traces.csv` file.

3.3.2 Feature Extraction

The extracted features was done based on each event sequence, resulting in dense vector per block. The process included

- **Counting events (Unigrams)** - Each event_id present in the sequence is counted. For example, if an event "E1" occurs three times, the event_E1 entry will have the value 3.
- **Counting bigrams (pairs of consecutive events)** - To capture sequential patterns, bigrams have been extracted, i.e. pairs of consecutive events. For example, a sequence ["E1", "E2", "E3"] generates the bigrams ("E1", "E2") and ("E2", "E3"). The count of these bigrams is stored in columns prefixed with bigram_.
- **Temporal metrics** - The TimeInterval column was used to calculate three aggregate statistics: Total sum of time (time_sum), average time between events (time_mean) and longest interval (time_max). These variables capture abnormal latency variations within the sequence.

The following listing illustrates the implementation of the feature extraction process:

```
def extract_features(row):
    events = row['Events']
    times = row['TimeInterval']
    # Event counts
    event_counts = Counter(events)
    # Bigrams
    bigrams = Counter([tuple(events[i:i+2]) for i in range(len(events)-1)])
    # Time aggregates
    time_sum = sum(times)
    time_mean = np.mean(times) if times else 0
    time_max = max(times) if times else 0
    # Combine features
    features = {f'event_{k}': v for k, v in event_counts.items()}
    features.update({f'bigram_{k}': v for k, v in bigrams.items()})
    features.update({
        'time_sum': time_sum,
        'time_mean': time_mean,
        'time_max': time_max,
        'latency': row['Latency'],
        'seq_length': len(events)
    })
    return features

# Apply feature extraction
if os.path.exists(os.path.join('variables', 'features_df.pkl')):
    features_df = pd.read_pickle(os.path.join('variables', 'features_df.pkl'))
else:
    feature_list = []
    for _, row in tqdm(df.iterrows(), total=len(df), desc="Extracting features"):
        feature_list.append(extract_features(row))
    features_df = pd.DataFrame(feature_list)
    features_df = features_df.fillna(0)  # Handle missing features # Handle missing features

    # Encode labels
    features_df['Label'] = (df['Label'].values == 'Fail').astype(int)  # 1 for Fail, 0 for Success
    features_df.to_pickle(os.path.join('variables', 'features_df.pkl'))
```

Listing 3.1: Feature extraction function.

3.3.3 Label Codification and Class Imbalance Handling

For each event, a label was binary encoded (0 or 1) depending on the original value (Success or Fail), respectively. However, as depicted in Figure 3.1, the dataset was highly imbalanced, with significantly more successful blocks than failed ones. This imbalance could bias the classifier toward the majority class. While this problem could be addressed using oversampling techniques such as Synthetic Minority Over-sampling Technique (SMOTE), which generates synthetic examples of the minority class, or undersampling, the contrary idea, in this project an alternative approach was adopted. Given the substantial size of the original dataset, the implementation of the `scale_pos_weight` parameter (available in all the covered models)

enabled the achievement of a comparable outcome by imposing penalties on the misclassification of the minority class, without needing an increase in the size of the training set. A typical value to consider, according to the XGBost documentation [48] and other sources is $sum(negativeinstances)/sum(positiveinstances)$ or 43.558, reflecting the relative proportion of the classes Additionally, Stratified K-Fold cross-validation was adopted during model training, ensuring that each fold preserved the original distribution of classes.

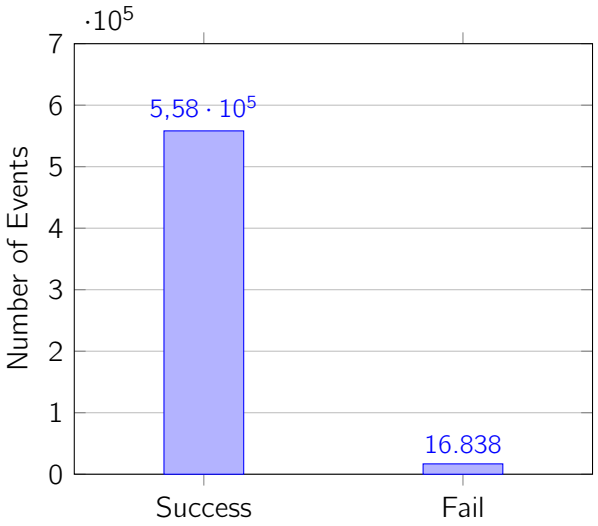


Figure 3.1: Class imbalance on original dataset

This is vital for obtaining reliable performance estimates and preventing bias towards the majority class during model evaluation. The objective of combining these techniques was to construct a robust and accurate model that could effectively predict system failures despite the challenges posed by imbalanced data, thereby maximising our ability to detect and prevent potential outages.

After data preprocessing and conducting an initial test fit using a `RandomForestClassifier` with no parameter tuning, it was possible to observe which features contributed most to the classification task, showing the importance of proper feature extraction and data pre processing. The top 10 were:

Table 3.1: Top 10 most important features ranked by their contribution

Feature	**Importance Score**
event_E20	0.1478
bigram_('E21', 'E20')	0.1100
seq_length	0.0807
bigram_('E21', 'E21')	0.0579
bigram_('E20', 'E21')	0.0575
event_E27	0.0418
bigram_('E26', 'E21')	0.0412
event_E26	0.0339
time_mean	0.0328
bigram_('E26', 'E27')	0.0245

These top 10 most important features, as identified by Random Forest, provide valuable insights into the factors contributing to system failures. The most important feature, event_E20, represents the frequency of log event E20, which corresponds to an Unexpected error trying to delete a block. This suggests that disk-related issues are a strong predictor of subsequent system instability. The importance of bigram_('E21', 'E20') further reinforces this, as it captures the co-occurrence of an attempt of deleting a block(E21) followed by the error (E20), further indicating failures from trying to delete block info. The presence of time_mean in the top features suggests that abnormal latency also plays a role, potentially indicating system slowdown or resource contention.

3.4 Modelling

The modelling stage concentrated on training and comparing different supervised learning algorithms for the binary classification of log sequences as either normal or failure-related. Four widely used models were selected on the basis of their proven performance in classification tasks and ability to handle high-dimensional data: Random Forest, XGBoost, LightGBM and CatBoost.

3.4.1 Model Selection

Random Forest [49] was chosen as a baseline ensemble method, leveraging multiple decision trees with bootstrap aggregation to reduce variance. Compared to single decision trees, Random Forest reduces over-fitting and improves generalization by averaging the predictions of multiple weak learners.

It served as a strong baseline because of its:

- Tolerance to irrelevant features and outliers
- Low requirement for hyper-parameter tuning
- Built-in feature importance computation
- Suitability for data with mixed feature types (e.g. event frequency, timing statistics)

Gradient Boosting Models

To try to achieve better results and improve upon the baseline, several gradient boosting models were also evaluated:

- **XGBoost:** a scalable, end-to-end Gradient Boosting Decision Tree (GBDT) that excels by processing billions of examples faster and with fewer resources than existing systems, while achieving state-of-the-art results, due to its algorithmic optimizations like a sparsity-aware algorithm for sparse data and a weighted quantile sketch for approximate learning, combined with system innovations focusing on cache access patterns, data compression, and sharding [50].
- **LightGBM:** a novel implementation of GBDT that significantly accelerates the training process by up to over 20 times and scales to high-dimensional and large-volume data while maintaining almost the same accuracy, primarily through its innovative techniques: Gradient-based One-Side Sampling (GOSS) for efficient data instance reduction and Exclusive Feature Bundling (EFB) for effective feature reduction [51].

- **CatBoost:** a gradient boosting toolkit that resolves the prediction shift caused by target leakage in existing GBDT implementations by incorporating ordered boosting and an innovative algorithm for processing categorical features using ordered target statistics [52].

3.4.2 Hyper-parameter Tuning

To optimize the model's performance, a hyperparameter tuning step was conducted using the Randomized Search approach (`RandomizedSearchCV`). This approach was chosen over the more exhaustive GridSearchCV due to the size of the dataset and the computational cost associated with evaluating all possible parameter combinations. While Grid Search explores the entire search space, Randomized Search samples a fixed number of parameter settings from specified distributions, significantly reducing computation time while still allowing for effective exploration of the hyperparameter space. For each model-XGBoost, LightGBM, and CatBoost—a custom parameter grid was defined. The search considered combinations of the number of estimators, tree depth, and learning rate, as presented in Table 3.2.

Table 3.2: Parameter grid used in RandomizedSearchCV

Parameter	**Values**
n_estimators	[100, 200, 300]
max_depth	[3, 5, 7]
learning_rate	[0.01, 0.1, 0.2]

The best parameter configurations identified for each model are shown in Table 3.3.

Table 3.3: Best Parameter grid values found in RandomizedSearchCV

Model	**Best Parameter Grid Values**
XGBoost	*n_estimators*: 300 *max_depth*: 7 *learning_rate*: 0.1
LightGBM	*n_estimators*: 300 *max_depth*: 7 *learning_rate*: 0.1
CatBoost	*n_estimators*: 300 *max_depth*: 5 *learning_rate*: 0.2

A 10-fold stratified cross-validation was used to ensure consistent class distribution in each split. The optimization objective was the macro F1 score, chosen to balance performance across both classes (normal and failure), given the imbalanced nature of the dataset.

3.5 Evaluation

All models achieved high performance, with macro F1-scores above 0.98, demonstrating their ability to accurately distinguish between normal and failure sequences. LightGBM slightly outperformed the others, achieving a macro F1-score of 0.9872, followed closely by XGBoost (0.9850) and CatBoost (0.9846). Precision and recall values for the minority class (failure) were particularly high across all models, with a 0.99 recall, indicating a low false-negative rate, which is crucial in failure prediction scenarios. A detailed breakdown of each model's performance, including precision, recall for the minority class, is presented in Table 3.4.

Table 3.4: Final performance of each model on the test set

Model	F1-macro	Precision (Fail)	Recall (Fail)
XGBoost	0.9850	0.9509	0.9913
LightGBM	0.9872	0.9612	0.9891
CatBoost	0.9846	0.9520	0.9885

As seen in Fig. 3.2, the two best-performing models are XGBoost and LightGBM.

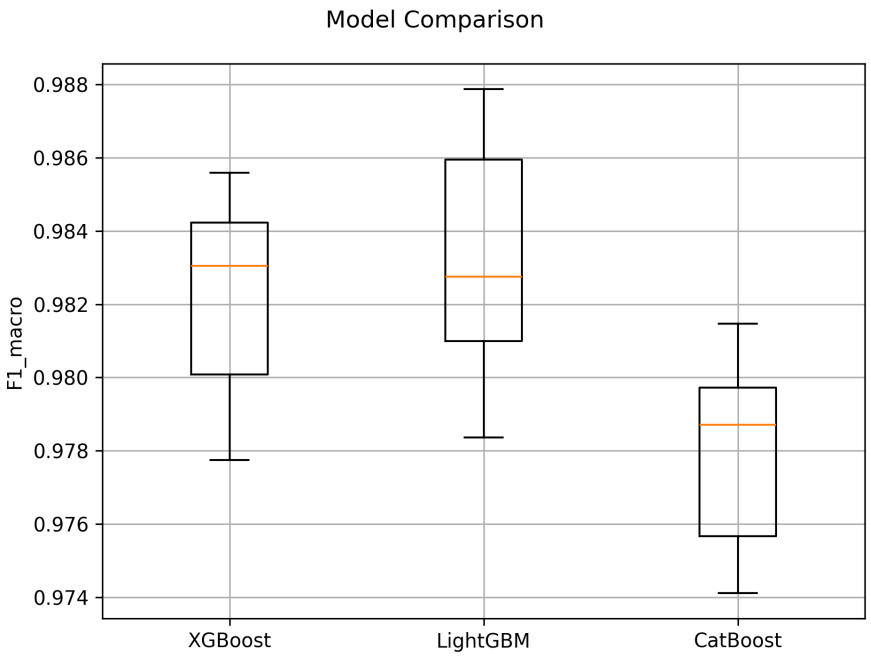


Figure 3.2: Models Comparison

3.5.1 Model Selection For Deployment

To compare the difference between the two best models more rigorously, a statistical test suite was carried out on the experimental results to validate their results further and determine whether there exists a significant difference among them.

To validate the hypothesis testing procedure described in 2.5.2, a Python function was implemented to automate the comparison between the two selected models. This function

first evaluates the normality of the fold-wise performance differences using the Shapiro-Wilk test. Depending on the outcome, it then performs either a paired t-test or a Wilcoxon signed-rank test to assess whether the performance difference is statistically significant. The implementation is presented below.

```
from scipy.stats import shapiro, ttest_rel, wilcoxon

def modelComparisonStatTests(model1, results_mod1, model2, results_mod2):
    differences = results_mod1 - results_mod2
    shapiro_test = shapiro(differences)
    print(f"\nShapiro-Wilk Test: Statistic = {shapiro_test.statistic:.4f}, p-value = {shapiro_test.pvalue:.4f}")

    alpha = 0.05
    if shapiro_test.pvalue > alpha:
        print("The differences are normally distributed (fail to reject H0).")
        t_test = ttest_rel(results_mod1, results_mod2)
        print(f"Paired t-test: Statistic = {t_test.statistic:.4f}, p-value = {t_test.pvalue:.4f}")
        if t_test.pvalue > alpha:
            print(f"No statistically significant difference between {model1} and {model2} (fail to reject H0).")
        else:
            print(f"Statistically significant difference between {model1} and {model2} (reject H0).")
    else:
        print("The differences are not normally distributed (reject H0).")
        wilcoxon_test = wilcoxon(results_mod1, results_mod2)
        print(f"Wilcoxon signed-rank test: Statistic = {wilcoxon_test.statistic:.4f}, p-value = {wilcoxon_test.pvalue:.4f}")
        if wilcoxon_test.pvalue > alpha:
            print(f"No statistically significant difference between {model1} and {model2} (fail to reject H0).")
        else:
            print(f"Statistically significant difference between {model1} and {model2} (reject H0).")
```

Listing 3.2: Statistical function to compare the performance of two models.

The function was executed to compare the XGBoost and LightGBM models based on their fold-wise F1-macro scores. The obtained results are shown bellow.

```
Shapiro-Wilk Test:  Statistic = 0.9030, p-value = 0.2361
The differences are normally distributed.
T-Test:  Statistic = -1.7831, p-value = 0.1082
There is no statistically significant difference between XGBoost and LightGBM.
```

Given this outcome, both models were selected for inclusion in the log streaming simulation, enabling a side-by-side comparison of their predictions in a real-time environment.

3.6 Deployment

The deployment phase focused on demonstrating how the trained models could be integrated into a real-time log monitoring and prediction pipeline. Whilst the work was conducted within a simulated environment, the design closely reflected the requirements of an operational distributed system.

3.6.1 Log Streaming Simulation

To simulate a realistic operational environment where logs are generated continuously, a custom log streaming script was developed. This component mimics the behaviour of a distributed system by sequentially emitting log event sequences in near real-time, allowing for continuous failure prediction using the trained model.

Objectives

The main goals of the streaming simulation were:

- Emulate the printing of log sequences over time, grouped by block_id
- Aggregate events as they arrive
- Apply the previously trained model to classify each sequence into Failure or Normal
- Output live predictions and simulate alerting behaviour

This setup enables an end-to-end evaluation of the failure prediction system under conditions closer to a production deployment, where decisions must be made incrementally and in real time.

3.6.2 Stream Emulation

The log stream was emulated using a generator function inspired by the Unix `tail -f` command. It continuously reads new lines appended to a file, simulating how logs are written by a live system. To allow the prediction system to run in parallel with log ingestion and avoid blocking the main process, the core prediction loop was executed in a separate thread.

```
def tail_f(file_path):
    with open(file_path, 'r') as f:
        f.seek(0)  # Start at the beginning of the file
        while True:
            line = f.readline()
            if not line:
                time.sleep(0.5)  # Wait before checking for new lines
                continue
            yield line
```

Listing 3.3: Log stream generator function.

Uppon arrival, logs are grouped dynamically by `block_id`:

```
log_buffer = {}

for _, row in df.iterrows():
    block_id = row['BlockId']
    event = row['EventId']
    time_diff = row['TimeInterval']

    # Simulate delay between log arrivals
    time.sleep(0.01)

    if block_id not in log_buffer:
        log_buffer[block_id] = {
            "events": [],
            "times": [],
            "latency": row["Latency"]
        }
    log_buffer[block_id]["events"].append(event)
    log_buffer[block_id]["times"].append(time_diff)
```

Listing 3.4: Function to group logs by block_id.

3.6.3 Event aggregation and eligibility

As new log entries arrive, they are grouped by `block_id`. However, not all sequences are immediately eligible for prediction. A configurable threshold is applied to ensure that only sequences with a sufficient number of events are evaluated by the model

```
MIN_EVENTS_THRESHOLD = 10

def should_predict(events):
    """Check if we have enough data to make a prediction."""
    return len(events) >= MIN_EVENTS_THRESHOLD
```

Listing 3.5: Prediction eligibility function.

3.6.4 Feature Building and Prediction

Once a sequence is considered eligible for prediction, typically after accumulating a minimum number of events, it is transformed into a fixed-size feature vector that captures both structural and temporal characteristics of the log trace. This transformation mimics the processing done at the data preparation phase where it involved computing the frequency of individual events (unigrams), sequential patterns (bigrams), and several aggregate timing metrics such as total duration, average time between events, and maximum interval. The feature extraction logic is detailed in Listing A.1 in Appendix A.

This vectorization step ensured that each log sequence, regardless of its length, is encoded into a consistent representation suitable for input into the trained classification model.

Once the features are extracted, they are aligned with the same schema used during the model's training phase to ensure consistency. The trained XGBoost and LightGBM models, previously optimized via cross-validation, were then used to predict whether the sequence is indicative of a failure or a normal execution path. The prediction step is shown in A.2 in Appendix A, and returns both the predicted label and its associated probability score.

This classification is performed in real time, immediately after a block's feature vector is completed, allowing for continuous, low-latency fault detection as logs arrive in the system.

Chapter 4

Log Streaming Simulation Results

This chapter presents the results obtained from embedding the trained machine learning models into a simulated log streaming environment. The objective of the study was to assess the predictive accuracy of the models, as well as their performance under real-time operational constraints.

4.1 Introduction

The log streaming simulation aimed to evaluate the real-time predictive performance of two selected models (XGBoost and LightGBM) when exposed to identical operational conditions. Both models processed the same sequential stream of aggregated log event windows, preserving the original temporal order of events from the dataset. This setup ensured that their predictions could be compared directly and without bias, as they were derived from the same input sequences and were processed in the same way. A fundamental aspect of this evaluation was the assessment of these models' capacity to minimize system downtime, a calculation that will be presented later in this chapter.

4.2 Key Findings

Figures 4.1 and 4.2 illustrate the predictive accuracy of XGBoost and LightGBM, respectively, throughout the log streaming simulation. Both models exhibit a characteristic learning curve, with an initial period of fluctuating performance that stabilizes as the number of predictions increases. XGBoost achieves a final Correct Predictions Ratio of approximately 0.973, while LightGBM reaches 0.905. This indicates that both models are generally effective at distinguishing between normal and failure sequences in a real-time streaming environment. The early volatility in performance suggests a potential 'warm-up' effect, where initial predictions are less reliable until the models adapt to the data stream. Overall, these figures demonstrate the feasibility of using both XGBoost and LightGBM for real-time failure prediction, with XGBoost exhibiting a slightly better, greater performance.

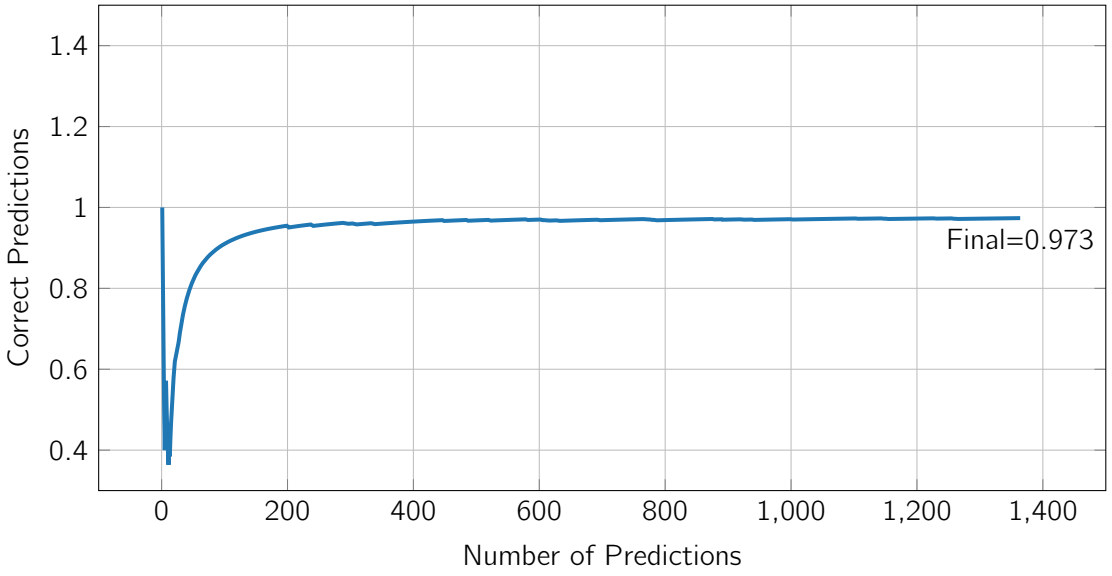


Figure 4.1: XGBoost Correct Predictions Ratio

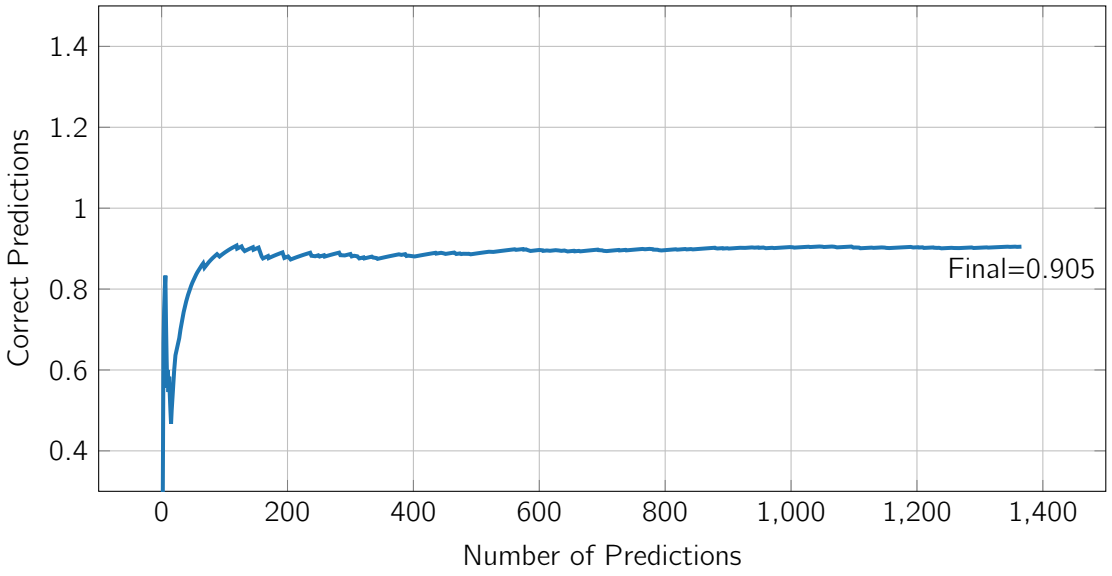


Figure 4.2: LightGBM Correct Predictions Ratio

In addition to these aggregate results, it is important to consider that certain blocks were evaluated more than once throughout the simulation. This occurs because a block's state may evolve over time, with new events providing additional context that can influence the prediction outcome. In some cases, an initially normal block was later flagged as indicative of failure. This highlights the importance of continuous re-evaluation for capturing the dynamic nature of distributed systems.

4.3 Inference Latency Analysis

In addition to evaluating the predictive performance of both models, the simulation measured the inference latency for each generated prediction. The time taken for classification output

was recorded for every processed log window. These measurements enabled the construction of an Empirical Cumulative Distribution Function (ECDF) for each model, offering insight into the proportion of predictions completed within specific time thresholds. The ECDF results demonstrate that both XGBoost and LightGBM consistently deliver inference times of less than a second, thus satisfying the real-time requirements of a failure prediction system. LightGBM exhibits marginally shorter average latency, though this difference is operationally negligible given the observed prediction frequencies.

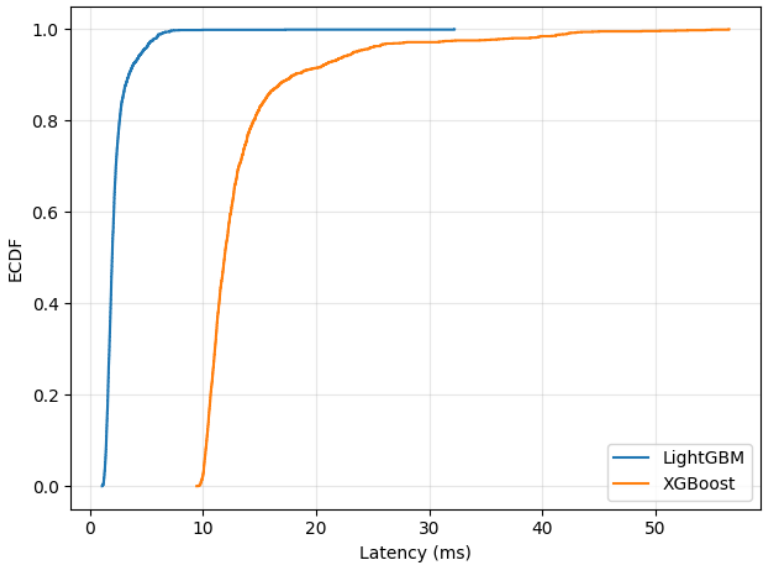
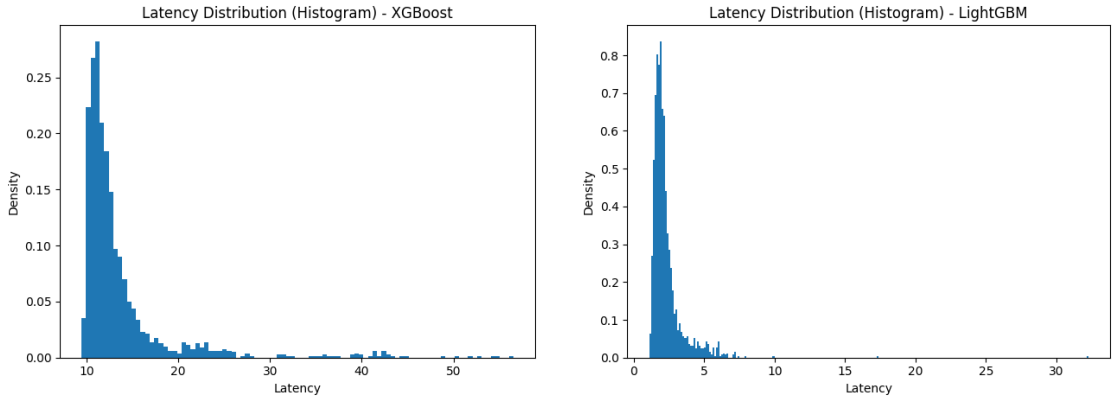


Figure 4.3: Log streaming ECDF

To complement the ECDF, Figures 4.4a and 4.4b show histograms of the per-prediction latency distribution for each model.



(a) XGBoost prediction latency (ms) histogram

(b) LightGBM prediction latency (ms) histogram

Figure 4.4: Models latency distribution histogram

These reveal that most predictions cluster tightly within a narrow range, indicating stable performance under continuous streaming. Most importantly, no significant latency spikes

were observed during the simulation, confirming that both models consistently respond well even when processing large sequences of events. While LightGBM exhibits a slight speed advantage in terms of inference latency, the performance of both models in terms of precision, recall, and F1-score was found to be comparable. Statistical testing revealed no significant difference between the predictive capabilities of the two models

4.4 State Transition Analysis (LightGBM vs. XGBoost)

In addition to measuring accuracy and latency, the log streaming simulation tracked how block states evolve across consecutive predictions. This analysis illustrates how model outputs adapt when the same block is reprocessed as new log events arrive. Table 4.1 summarises these state transitions for both LightGBM and XGBoost, categorising them as follows:

- $N \rightarrow N$ – block remained classified as normal;
- $N \rightarrow F$ – block transitioned from normal to failure;
- $F \rightarrow F$ – block transitioned from failure to normal;
- $F \rightarrow F$ – block remained classified as failure.

The $N \rightarrow F$ rate is the proportion of initially normal blocks that later became failures, and the $F \rightarrow N$ rate is the proportion of blocks labelled as failures that later became normal again.

Table 4.1: Comparison of block state transitions for LightGBM and XGBoost

Model	N→N	N→F	F→N	F→F	N→F rate	F→N rate
LightGBM	4	6	5	6	0.600	0.455
XGBoost	3	5	5	9	0.625	0.357

These results demonstrate that LightGBM is more flexible in reclassifying blocks from failure back to normal, as reflected by its higher $F \rightarrow N$ rate (45.5%). However, this behaviour may result in ongoing problems being dismissed prematurely. By contrast, XGBoost is more conservative, maintaining a failure state more persistently (with a higher $F \rightarrow F$ count of 9). This helps to detect prolonged issues, but can lead to more false positives if conditions improve.

This divergence raises the question of which behaviour is preferable in practice. A flexible approach, such as that of LightGBM, can be beneficial in environments where transient anomalies are common and systems recover quickly. By reclassifying blocks back to a standard state at an earlier point in the process, it is possible to reduce the number of unnecessary alerts and avoid the need for operator intervention in situations where the issue would have otherwise resolved itself. However, it should be noted that there is a risk of overlooking genuine failures if they are too quickly dismissed, which would result in less time being available to implement preventive measures.

By contrast, a conservative approach, as exemplified by XGBoost, ensures that once a failure is predicted, it remains flagged more persistently. This behaviour reduces the likelihood of missing ongoing issues and is especially valuable in mission-critical contexts where the cost of

undetected failures is greater than the nuisance of false alarms. Nevertheless, such caution can result in an elevated risk of generating false positives, a scenario that has the potential to culminate in "alert fatigue" and, by extension, a reduction in the level of trust placed in the system.

The implications of such behaviours are dependent upon the operational context. In highly resilient infrastructures, flexibility may be preferable in order to maintain agility and reduce unnecessary escalations. Conversely, in safety-critical systems, a conservative approach has the advantage of providing stronger assurance, despite the potential for more frequent alerts. It can be concluded that there is no inherent superiority of either strategy; the trade-off between flexibility and caution must be aligned with system requirements, risk tolerance, and operational priorities.

Incorporating additional gating factors, such as prediction confidence, recent state stability or agreement between multiple models, could allow predictions for borderline cases to be deferred until more evidence accumulates, while still issuing early warnings for high-confidence detections. This fine-tuning could reduce false positives and premature classifications, thus improving the stability of real-time predictions without compromising responsiveness.

Overall, these experiments confirm that, with some continuous tuning in the pipeline, both models can sustain accurate, low-latency predictions under realistic streaming conditions. This fulfils the operational requirements for proactive failure detection in distributed systems.

4.5 Downtime Avoidance Analysis

Beyond the measurement of predictive accuracy and latency, the streaming simulation was expanded to evaluate the practical impact of the models on system downtime. The objective of this analysis was to determine the extent to which service unavailability could be theoretically prevented if the predictions generated by the models were acted upon in advance.

The approach required several additional mechanisms to be implemented in the simulation code:

- **Proxy failure time** - Given the absence of explicit timestamps for failure occurrences in the HDFS dataset, the last observed log message for each failing block was utilized as a proxy for the failure instant. This assumption is consistent with the dataset structure, as blocks labelled as failures terminate with events linked to the failure sequence.
- **First failure prediction** - During the streaming process, the simulation captured the timestamp corresponding to the initial instance in which the model predicted a block as failing. This timestamp, derived from the log event times, represents the earliest possible warning that the model could provide for that block.
- **Lead time** - The lead time was calculated as the temporal difference between the proxy failure time and the first failure prediction. In the case of an earlier prediction, the lead time is positive; in the case of a later prediction, the lead time is zero or negative.
- **Mitigation threshold** - In order to operationalise the concept of downtime avoidance, a mitigation threshold parameter (`MITIGATION_MIN`) was introduced. This threshold indicates the minimum advance notice required for an operator or automated recovery

mechanism to successfully act on a prediction. In this evaluation, a threshold of five minutes was applied, representing a conservative assumption aligned with manual or semi-manual intervention workflows.

- **Downtime calculation** - Each failing block was associated with a baseline downtime of ten minutes. In the event that a model's prediction occurred a minimum of five minutes prior to the proxy failure time, the downtime associated with that particular block was considered to have been fully avoided. Predictions that occurred too late, or blocks that were never predicted as failures, were classified as non-avoided. The total duration of avoided downtime across all failing blocks was then compared against the baseline to calculate the percentage reduction in downtime:

$$Reduction\ (\%) = \frac{D_{base} - D_{ML}}{D_{base}} \times 100$$

where D_{base} is the baseline downtime and D_{ML} is the residual downtime after considering predictions.

The implementation of these mechanisms required the simulation to track, for each block, the last event timestamp, the first prediction of failure, and the classification of each failure into one of four categories: fully avoided, partially avoided, late prediction, or not predicted. Aggregated results were then exported as CSV files containing both detailed per-block breakdowns and summary statistics.

4.5.1 XGBoost Results

The initial extensive run of 10,000 predictions using XGBoost processed 9,981 unique blocks, of which 252 were labelled as failures in the ground truth. This subset corresponds to approximately 1.5% of all failing blocks in the dataset, a proportion that is referred to as coverage. In this context, coverage indicates the proportion of ground-truth failures that were actually observed within the processed slice of the log stream. Therefore, coverage represents the maximum number of failures that the model could possibly anticipate in that run. Within the set of 252 failing blocks, nine were identified as being at risk of failure, with a sufficient lead time to meet the five-minute mitigation threshold. The findings of the study indicated that the total number of minutes of avoided downtime was equivalent to 90 minutes out of a baseline of 2,520 minutes. This calculation yielded a percentage reduction in downtime of 3.6%.

- Blocks processed - 9,981
- Failing Blocks - 252
- True Positives - 9
- Baseline Downtime (min) - 2,520
- Downtime avoided (min) - 90
- Reduction (%) - 3.6
- Coverage (%) - 1.5

Despite their modest appearance, these results demonstrate the capacity of XGBoost to generate actionable early warnings under streaming conditions. The relatively low reduction percentage can be attributed to two factors: the strict five-minute mitigation

requirement, which excluded predictions made only slightly before the proxy failure instant, and the limited coverage (1.5% of failing blocks), since only a portion of the dataset was processed in the evaluated run. Nevertheless, the presence of non-zero avoided downtime highlights the potential operational value of the model.

4.5.2 LightGBM Results

Applying the same methodology to LightGBM produced substantially stronger results. Over an equivalent run of 10 000 predictions, the model again processed 9,981 unique blocks, including 252 failing blocks in the ground truth. Out of these, 99 failing blocks were predicted as failures early enough to satisfy the five-minute mitigation condition. This translated into 990 minutes of avoided downtime out of the same baseline of 2,520 minutes, corresponding to a 39.3% reduction in downtime.

- Blocks processed - 9,981
- Failing Blocks - 252
- True Positives - 99
- Baseline Downtime (min) - 2,520
- Downtime avoided (min) - 990
- Reduction (%) - 39.3
- Coverage (%) - 1.5

The contrast with XGBoost is significant: LightGBM demonstrated a capacity to prevent failures that was more than ten times greater than XGBoost, thereby achieving a reduction in downtime that was one order of magnitude higher. This finding indicates that LightGBM's learned decision boundaries exhibited enhanced alignment with early indicators of failure in the log sequences, enabling it to satisfy the mitigation condition with greater frequency. In practice, this would result in a significantly higher proportion of actionable early warnings.

4.5.3 Comparative Reflection

The downtime avoidance analysis reveals significant differences between the two models. While both systems demonstrated high levels of predictive accuracy in offline evaluations, a significant divergence in their effectiveness in producing timely warnings was observed under the operational constraints imposed in the streaming scenario.

- XGBoost demonstrated a certain degree of efficacy in predicting failures; however, the total amount of avoided downtime remained comparatively low (3.6%). This outcome is indicative of its more conservative prediction behaviour, as observed in the transition analysis, where it exhibited a tendency to maintain failure states persistently but demonstrated a reduced rate of reclassification of blocks as failing.
- LightGBM exhibited a substantially greater capacity to detect failures in advance, thereby avoiding nearly 40% of downtime in the evaluated slice. Its behaviour can be characterised as more flexible and adaptive, reclassifying blocks earlier in the presence of failure-indicative patterns.

From an operational perspective, the results emphasise the necessity of evaluating predictive models not only on accuracy but also on timeliness and practical impact. The value of a prediction is negligible, regardless of its accuracy, if it occurs after or only moments before a failure. Conversely, a prediction that provides several minutes of lead time has the capacity to directly reduce downtime, provided that the environment supports rapid mitigation.

It is imperative to emphasise that the results presented here are based on a partial coverage of the dataset (1.5% of failing blocks), given the immense size of the dataset, combined with the considerable time needed to run each prediction pipeline end to end. Broader evaluations or alternative mitigation thresholds (e.g., one or two minutes in automated recovery contexts) may alter the relative balance between the models. However, under the assumption of a strict five-minute assumption employed in this study, LightGBM exhibited a pronounced advantage in converting predictive capability into tangible reductions in downtime.

Chapter 5

Conclusions and Future Work

This dissertation presented the design, implementation and evaluation of a log analysis pipeline for predicting failures in real time in distributed systems. The study combined feature engineering from structured log sequences, machine learning model training and log streaming simulation to evaluate predictive performance under realistic operational conditions.

Offline evaluation showed that XGBoost and LightGBM both achieved high predictive accuracy, with macro-F1 scores exceeding 0.98 on test data. Statistical hypothesis testing confirmed that the observed differences in performance between the two best models were not statistically significant, suggesting that either model could be a viable option for deployment.

Extending the evaluation to a sequential, real-time setting via log streaming simulation provided insights into prediction stability, state transition behaviour, latency performance, and downtime avoidance analysis. The results showed that both models maintained high correct prediction ratios throughout the simulation, with disagreements primarily arising during an initial period, but they stabilised as the number of predictions increased. Analysis of block state transitions revealed differences in how each model handled persistence of failure states, highlighting the trade-off between caution and flexibility. Latency analysis, supported by ECDF and histogram visualisations, confirmed that both models consistently met real-time inference requirements, with the vast majority of predictions being completed in under one second. Furthermore, the stability of latency throughout the simulation indicated robustness under sustained event processing loads. Downtime avoidance analysis revealed that, under the assumed five-minute mitigation threshold, LightGBM was able to help in reducing system downtime by 39.3%.

While this thesis has successfully demonstrated the feasibility of real-time failure prediction in distributed systems using machine learning and log analysis, certain objectives outlined in Chapter 1 were not fully achieved within the scope of this work. Specifically, the design of a generalisable system architecture and the development of a user interface for alert exploration were not implemented. Furthermore, the model's performance was evaluated using a simulated environment based on the HDFS dataset from LogHub, and validation with real-world data from production systems remains an area for future research.

To address these gaps, future work should focus on: designing a more generalizable system architecture to support various distributed systems and log formats, developing a user-friendly interface to explore alerts and supporting logs, and conducting real-world validation with production data to assess the pipeline's performance in a realistic setting.

In conclusion, despite these limitations, this thesis demonstrates the feasibility and practicality of using machine learning for predictive maintenance in distributed systems when structured log sequences are the primary data source. Dual evaluation of XGBoost and LightGBM provided complementary insights into model behaviour, latency performance and decision stability, thereby reinforcing the value of comparative experimentation in deployment-oriented research. By combining data preparation, statistical validation and real-time simulation within a coherent methodology, the study provides a functional prototype and a systematic approach that can be applied to other large-scale, event-driven environments. These results pave the way for future research into adaptive ensemble methods, confidence-based decision mechanisms and operational monitoring strategies, which could enhance the reliability and responsiveness of predictive systems in production contexts even further.

Bibliography

[1] David Reinsel, John Gantz, and John Rydning. "The Digitization of the World From Edge to Core". In: *International Data Corporation* (Nov. 2018).

[2] Schleier-Smith and Johann Markus. "Understanding and Exploring Serverless Cloud Computing - ProQuest". In: *University of California, Berkeley-ProQuest Dissertations & Theses* (2022). url: https://www.proquest.com/openview/59fff697de5317f873d317e9342d7f1?pq-origsite=gscholar%5C&cbl=18750%5C&diss=y.

[3] Adam Oliner, Archana Ganapathi, and Wei Xu. "Advances and challenges in log analysis". In: *Communications of the ACM* 55 (2 Feb. 2012), pp. 55–61. issn: 00010782. doi: 10.1145/2076450.2076466.

[4] Taneja Rohan. *Biggest IT outages from 2023-2025*. Aug. 2025. url: https://zenduty.com/blog/it-outages/.

[5] *The True Cost of Website Downtime in 2025 |SiteQwality*. May 2025. url: https://siteqwality.com/blog/true-cost-website-downtime-2025/.

[6] Cole Stryker and Eda Kavlakoglu. *What Is Artificial Intelligence (AI)? |IBM*. Aug. 2024. url: https://www.ibm.com/topics/artificial-intelligence.

[7] Zhuangbin Chen et al. "A Survey on Automated Log Analysis for Reliability Engineering". In: *2021. A Survey on Automated Log Analysis for Reliability Engineering. ACM Comput. Surv* 1 (2021), p. 37. doi: 10.1145/1122445.1122456. url: https://doi.org/10.1145/1122445.1122456.

[8] Rhoda Ajayi. "Integrating IoT and Cloud Computing for Continuous Process Optimization in Real Time Systems". In: *International Journal of Research Publication and Reviews* 6 (Aug. 2025), pp. 2540–2558. doi: 10.55248/gengpi.6.0125.0441.

[9] Nick Hotz. *What is CRISP DM? - Data Science PM*. Nov. 2024. url: https://www.datascience-pm.com/crisp-dm-2/.

[10] Jeff Saltz. *CRISP-DM is Still the Most Popular Framework for Executing Data Science Projects - Data Science PM*. Nov. 2024. url: https://www.datascience-pm.com/crisp-dm-still-most-popular/.

[11] IBM Corporation. *CRISP-DM Help Overview - Documentação da IBM*. Aug. 2021. url: https://www.ibm.com/docs/pt-br/spss-modeler/saas?topic=dm-crisp-help-overview.

[12] Maarten van Steen and Andrew S. Tanenbaum. "A brief introduction to distributed systems". In: *Computing* 98 (10 Oct. 2016), pp. 967–1009. issn: 0010485X. doi: 10.1007/S00607-016-0508-7/FIGURES/13. url: https://link.springer.com/article/10.1007/s00607-016-0508-7.

[13] Arif Sari et al. "Fault Tolerance Mechanisms in Distributed Systems". In: *International Journal of Communications, Network and System Sciences* 8 (12 Dec. 2015), pp. 471–482. issn: 1913-3715. doi: 10.4236/IJCNS.2015.812042. url: https://www.scirp.org/journal/paperinformation?paperid=61986%20https://www.scirp.org/journal/paperabs?paperid=61986.

[14] Peronto Riley. *The State of Log Data: 6 Trends*. Oct. 2024. url: https://chronosphere.io/learn/observability-log-data-trends/.

[15] Shashi Shekhar Kumar and Sonali Agarwal. "Rule based complex event processing for IoT applications: Review, classification and challenges". In: *Expert Systems* 41 (9 Sept. 2024), e13597. issn: 14680394. doi: 10.1111/EXSY.13597;WGROUP:STRING:PUBLICATION. url: /doi/pdf/10.1111/exsy.13597%20https://onlinelibrary.wiley.com/doi/abs/10.1111/exsy.13597%20https://onlinelibrary.wiley.com/doi/10.1111/exsy.13597.

[16] *Fluentd |OpenSourceDataCollector|UnifiedLoggingLayer*. url: https://www.fluentd.org/.

[17] *Logstash: Collect, Parse, Transform Logs |Elastic*. url: https://www.elastic.co/logstash.

[18] *Collector |OpenTelemetry*. url: https://opentelemetry.io/docs/collector/.

[19] Jake Donnell. *Fundamentals of a Successful Logging and Observability Strategy |Logz.io*. Aug. 2024. url: https://logz.io/blog/logging-and-observability.

[20] Sara Brown. *Machine learning, explained |MITSloan*. Apr. 2021.

[21] *What Is Machine Learning (ML)? |IBM*.

[22] Shan Ali et al. "A comprehensive study of machine learning techniques for log-based anomaly detection". In: *Empirical Software Engineering* 30 (5 Oct. 2025), pp. 1–59. issn: 15737616. doi: 10.1007/S10664-025-10669-3/FIGURES/12. url: https://link.springer.com/article/10.1007/s10664-025-10669-3.

[23] Vannel Zeufack et al. "An unsupervised anomaly detection framework for detecting anomalies in real time through network system's log files analysis". In: *High-Confidence Computing* 1 (2 Dec. 2021), p. 100030. issn: 2667-2952. doi: 10.1016/J.HCC.2021.100030. url: https://www.sciencedirect.com/science/article/pii/S2667295221000209.

[24] *GitHub - logpai/logparser: A machine learning toolkit for log parsing [ICSE'19, DSN'16]*. url: https://github.com/logpai/logparser.

[25] Pinjia He et al. "Drain: An Online Log Parsing Approach with Fixed Depth Tree". In: *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017* (Sept. 2017), pp. 33–40. doi: 10.1109/ICWS.2017.13.

[26] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. "A lightweight algorithm for message type extraction in system application logs". In: *IEEE Transactions on Knowledge and Data Engineering* 24 (11 2012), pp. 1921–1936. issn: 10414347. doi: 10.1109/TKDE.2011.138.

[27] Risto Vaarandi and Mauno Pihelgas. "LogCluster-A Data Clustering and Pattern Mining Algorithm for Event Logs". In: ISBN (2015), pp. 978–981. url: http://dl.ifip.org/db/conf/cnsm/cnsm2015/1570161213.pdf.

[28] *GitHub - logpai/loghub: A large collection of system log datasets for AI-driven log analytics [ISSRE'23]*. url: https://github.com/logpai/loghub.

[29] Wael Khreich et al. "An anomaly detection system based on variable N-gram features and one-class SVM". In: *Information and Software Technology* 91 (Nov. 2017), pp. 186–197. issn: 09505849. doi: 10.1016/J.INFSOF.2017.07.009. url: https://www.researchgate.net/publication/318646668%5C%5FAn%5C%5FAnomaly%5C%5FDetection%5C%5FSystem%5C%5Fbased%5C%5Fon%5C%5FVariable%5C%5FN-gram%5C%5FFeatures%5C%5Fand%5C%5FOne-Class%5C%5FSVM.

[30] Zhong Li and Matthijs van Leeuwen. "Feature Selection for Fault Detection and Prediction based on Event Log Analysis". In: *ACM SIGKDD Explorations Newsletter* 24 (2 Aug. 2022), pp. 96–104. doi: 10.1145/3575637.3575652. url: http://arxiv.org/abs/2208.09440%20http://dx.doi.org/10.1145/3575637.3575652.

[31] Weibin Meng et al. "A Semantic-aware Representation Framework for Online Log Analysis". In: (). url: https://github.com/WeibinMeng/Log2Vec.

[32] Sasho Nedelkoski et al. "Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs". In: *Proceedings - IEEE International Conference on Data Mining, ICDM* 2020-November (Aug. 2020), pp. 1196–1201. issn: 15504786. doi: 10.1109/ICDM50108.2020.00148. url: https://arxiv.org/pdf/2008.09340.

[33] *Full-Stack Observability & Security Built for Enterprise Scale |Datadog*. url: https://www.datadoghq.com/dg/monitor/free-trial/?utm%5C%5Fsource=google&utm%5C%5Fmedium=paid-search&utm%5C%5Fcampaign=dg-brand-ww&utm%5C%5Fkeyword=datadog&utm%5C%5Fmatchtype=b&igaag=95325237782&igaat=&igacm=9551169254&igacr=673270769690&igakw=datadog&igamt=b&igant=g&utm%5C%5Fcampaignid=9551169254&utm%5C%5Fadgroupid=95325237782&gad%5C%5Fsource=1&gad%5C%5Fcampaignid=9551169254.

[34] *Splunk |TheKeytoEnterpriseResilience*. url: https://www.splunk.com/.

[35] Datadog. *Log Management*. url: https://docs.datadoghq.com/logs/.

[36] Datadog. *Data intake*. url: https://docs.datadoghq.com/partners/getting%5C%5Fstarted/data-intake/.

[37] Sematex. *Datadog vs Splunk: Comparison, key features, and overview*. url: https://sematext.com/blog/datadog-vs-splunk/.

[38] BetterStack. *Datadog vs. Splunk: a side-by-side comparison for 2025 |BetterStackCommunity*. url: https://betterstack.com/community/comparisons/datadog-vs-splunk/.

[39] Chrissy Kidd. *What Is Splunk? The Complete Overview of What Splunk Does |Splunk*. June 2025. url: https://www.splunk.com/en%5C%5Fus/blog/learn/what-splunk-does.html.

[40] Yuxia Xie, Kai Yang, and Pan Luo. "LogM: Log Analysis for Multiple Components of Hadoop Platform". In: *IEEE Access* 9 (2021), pp. 73522–73532. issn: 21693536. doi: 10.1109/ACCESS.2021.3076897.

[41] Yanjie Sun, Yali Gao, and Xiaoyong Li. "OptimizeLog: Log Anomaly Detection and Localization based on Optimized Log Parsing in Distributed Systems". In: *2023 9th International Conference on Computer and Communications, ICCC 2023*. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 2144–2148. doi: 10.1109/ICCC59590.2023.10507473.

[42] Dai Zaojian et al. "Semi-supervised Power Microservices Log Anomaly Detection Based on BiLSTM and BERT with Attention". In: *Proceedings - 2023 2nd International Conference on Advanced Electronics, Electrical and Green Energy, AEEGE 2023*. Institute of Electrical and Electronics Engineers Inc., 2023, pp. 82–87. doi: 10.1109/AEEGE58828.2023.00023.

[43] Nicolas Aussel, Yohan Petetin, and Sophie Chabridon. "Improving performances of log mining for anomaly prediction through nlp-based log parsing". In: *Proceedings - 26th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2018*. Institute of Electrical and Electronics Engineers Inc., Nov. 2018, pp. 237–243. doi: 10.1109/MASCOTS.2018.00031.

[44] Tao Li et al. "FLAP: An end-to-end event log analysis platform for system management". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Vol. Part F129685. Association for Computing Machinery, Aug. 2017, pp. 1547–1556. doi: 10.1145/3097983.3098022.

[45] Weibin Meng et al. "Loganomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs". In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. IJCAI'19. Macao, China: AAAI Press, 2019, pp. 4739–4745. isbn: 9780999241141.

[46] Wei Xu et al. "Detecting large-scale system problems by mining console logs". In: *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), pp. 117–131. doi: 10.1145/1629575.1629587. url: /doi/pdf/10.1145/1629575.1629587?download=true.

[47] Jieming Zhu et al. "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics". In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (Aug. 2020), pp. 355–366. issn: 10719458. doi: 10.1109/ISSRE59848.2023.00071. url: https://arxiv.org/pdf/2008.06448.

[48] xgboost developers. *XGBoost Parameters – xgboost 3.1.0-dev documentation*. url: https://xgboost.readthedocs.io/en/latest/parameter.html.

[49] Leo Breiman. "Random forests". In: *Machine Learning* 45 (1 Oct. 2001), pp. 5–32. issn: 08856125. doi: 10.1023/A:1010933404324/METRICS. url: https://link.springer.com/article/10.1023/A:1010933404324.

[50] Tianqi Chen and Carlos Guestrin. "XGBoost: A scalable tree boosting system". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 13-17-August-2016 (Aug. 2016), pp. 785–794. doi: 10.1145/2939672.2939785/SUPPL_FILE/KDD2016_CHEN_BOOSTING_SYSTEM_01-ACM.MP4. url: https://dl.acm.org/doi/pdf/10.1145/2939672.2939785.

[51] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems* 30 (2017). url: https://github.com/Microsoft/LightGBM..

[52] Liudmila Prokhorenkova et al. "CatBoost: unbiased boosting with categorical features". In: *Advances in Neural Information Processing Systems* 31 (2018). url: https://github.com/catboost/catboost.

Appendix A

Code Listings

```python
def build_feature_vector(events: list[tuple[str, float]],
    expected_columns: list[str]) -> pd.DataFrame:
    """
    Build a full feature vector from a list of (event_id, timestamp)
    tuples.
    Output is a DataFrame with a single row aligned to the model input.
    """

    event_counts = defaultdict(int)
    bigram_counts = defaultdict(int)
    timestamps = []

    for event_id, timestamp, _ in events:
        event_counts[f"event_{event_id}"] += 1
        timestamps.append(timestamp)

    # Create bigrams
    for (e1, _, _), (e2, _, _) in pairwise(events):
        bigram = f"bigram_{e1}_{e2}"
        bigram_counts[bigram] += 1

    # Time metrics
    if timestamps:
        latency = timestamps[-1] - timestamps[0]
        time_diffs = [t2 - t1 for t1, t2 in zip(timestamps[:-1],
    timestamps[1:])]
        time_sum = sum(time_diffs)
        time_mean = sum(time_diffs) / len(time_diffs) if time_diffs else
     0
        time_max = max(time_diffs) if time_diffs else 0
    else:
        latency = time_sum = time_mean = time_max = 0

    seq_length = len(events)

    # Merge all features
    all_features = {
        **event_counts,
        **bigram_counts,
        "time_sum": time_sum,
        "time_mean": time_mean,
        "time_max": time_max,
        "latency": latency,
        "seq_length": seq_length
    }

    # Align with model's expected input
    row = {col: all_features.get(col, 0) for col in expected_columns}
    return pd.DataFrame([row])
```

Listing A.1: Feature vector extraction function.

```
from xgboost import XGBClassifier

# Load trained model
if model_name == 'XGBoost':
    model: XGBClassifier = joblib.load(os.path.join("models", "
    XGBoost_model.pkl"))
elif model_name == 'LightGBM':
    model: LGBMClassifier = joblib.load(os.path.join("models", "
    LightGBM_model.pkl"))

model_row = build_feature_vector(events, expected_columns)
prediction = model.predict(model_row)
prediction_proba = model.predict_proba(model_row)

true_label = ground_truth.get(block_id)

if true_label is not None:
    is_correct = (prediction[0] == true_label)
    total_predictions += 1
    correct_predictions += int(is_correct)
    print(f"{Fore.YELLOW}Ground truth: {'Failure' if true_label == 1
    else 'Normal'} | "
          f"{'Correct' if is_correct else 'Incorrect'}")
else:
    print(f"{Fore.LIGHTBLACK_EX}No ground truth available for Block {
    block_id}")
```

Listing A.2: Sequence prediction logic.

Appendix B

Tables

BlockId	Label	Type	Features	TimeInterval	Latency
blk_7854771516489510256	Success		[E5, E5, E22, E5, E11, E9, E11, E9, ...]	[0.0, 0.0, 1.0, 48.0, 0.0, 0.0, 0.0, 0.0, ...]	50583
blk_-8531310335568756456	Fail	4	[E5, E22, E5, E5, E11, E9, E11, E9, ...]	[0.0, 2.0, 0.0, 56.0, 0.0, 0.0, 0.0, 0.0, ...]	51107
blk_-3409923645141256069	Success		[E5, E22, E5, E5, E11, E9, E11, E9, ...]	[0.0, 1.0, 9.0, 55.0, 0.0, 0.0, 0.0, 0.0, ...]	50443
blk_3947106522258141922	Fail	4	[E5, E5, E22, E5, E11, E9, E11, E9, ...]	[0.0, 0.0, 1.0, 27.0, 0.0, 0.0, 0.0, 0.0, ...]	51509
blk_-6535155942125829332	Success		[E5, E5, E5, E22, E11, E9, E11, E9, ...]	[0.0, 0.0, 0.0, 27.0, 0.0, 0.0, 0.0, 0.0, ...]	50528

Table B.1: Event_Traces.csv data sample