Automated Log Analysis: Failure Prediction and Anomaly Detection using Machine Learning and Large Language Models

by

Fatemeh Hadadi

Thesis submitted to the
Faculty of Engineering
In partial fulfillment of the requirements
For the Ph.D. degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

The dependability of modern software systems is becoming increasingly crucial as their complexity and scope continue to grow. Log data recorded during system execution can be leveraged to predict failures and detect anomalies automatically. However, designing accurate and efficient log-based analysis methods remains challenging due to the diversity of system environments, the instability of logs over time, and the scarcity of labeled data for training. This thesis addresses these challenges by systematically evaluating deep learning models for log-based failure prediction and by proposing a hybrid data-efficient approach that combines machine learning and a Large Language Model (LLM) for anomaly detection in unstable logs.

The first part of the thesis focuses on failure prediction using log data. While several Machine Learning (ML) and Deep Learning (DL) methods have been proposed, existing empirical studies are limited in scope, often examining only a subset of DL architectures or narrow dataset conditions. This thesis systematically investigates the combination of different log embedding strategies with major types of DL architectures, including Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and transformers. To enable a comprehensive evaluation, we design a modular architecture to accommodate various embedding strategies with different DL encoder configurations and synthesize 360 datasets with controlled characteristics, such as dataset size and failure rate, across three distinct system behavioral models. Experimental results demonstrate that CNN-based models with the Logkey2vec embedding strategy achieve the best overall performance, particularly when the dataset size exceeds 350 instances or when the failure rate exceeds 7.5%. These findings provide actionable insights into selecting effective models depending on dataset characteristics.

The second part of the thesis addresses Anomaly Detection in Unstable Logs (ULAD), a more realistic but underexplored situation where logs evolve due to software or environmental changes. Existing approaches based on machine learning typically require substantial labeled data, whereas LLMs can generalize with little data but struggle to capture structured log patterns. To address these complementary limitations, this thesis introduces FlexLog, a novel hybrid approach that integrates simple ML models (Decision Trees (DT), K-Nearest Neighbors (KNN), and Single-layer Feedforward Network (SLFN)) with a LLM (Mistral) through ensemble learning. FlexLog further incorporates a cache and a retrieval-augmented generation (RAG) components to improve time efficiency and accuracy, respectively. We evaluate FlexLog on four datasets specifically configured for unstable log anomaly detection. Results show that FlexLog consistently outperforms baseline methods by at least 1.2 percentage points (pp) in F1 score, while reducing labeled data requirements by 62.87 pp. When trained on the same amount of data as the baselines, FlexLog achieves up to a 13 pp increase in F1 score on the ADFA-U dataset, while maintaining an inference time of less than one second per log sequence, making it suitable for most practical applications.

By systematically evaluating deep learning architectures for log-based failure prediction and introducing a hybrid ML-LLM framework for detecting unstable log anomalies, this thesis provides a unified and empirical foundation for advancing log-based dependability

analysis. To ensure reproducibility and enable future research, all datasets, tools, and experimental results are made publicly available.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Canada Research Chairs and Discovery Grant programs, by the University of Luxembourg's joint research program grant, by the Luxembourg National Research Fund (FNR) under grant C22/IS/17373407/LOGODOR, by Science Foundation Ireland under Grant 13/RC/2094-2, and by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254 (COSMOS). The experiments conducted in this work were enabled in part by computation support provided by the Digital Research Alliance of Canada [1].

I am profoundly grateful to my supervisor, Prof. Lionel Briand, for his exceptional guidance, encouragement, and vision throughout my PhD. His mentorship has been pivotal in shaping both this thesis and my development as a researcher.

I would also like to thank Prof. Domenico Bianculli for his insightful advice and constructive feedback, which consistently refined my research. I am deeply thankful to Prof. Donghwon Shin, whose expertise in log analysis informed key aspects of the algorithms developed in this thesis. I am also grateful for my postdoctoral mentors, Dr. Joshua Dawes and Dr. Qinghua Xu, for their steady support and insightful input at every stage of this journey.

I am thankful to the members of my Thesis Committee, Prof. Diana Inkpen, Prof. Olga Baysal, Prof. Paula Branco, and Prof. Mika Mäntylä, for their valuable feedback and time.

Finally, I also thank my colleagues in the Nanda Lab, who offered encouragement and help during different, sometimes challenging stages of my PhD, making the process more rewarding.

Dedication

First and foremost, this dissertation is dedicated to my parents, whose unwavering support has carried me across both oceans and deadlines. From my lunch-time calls that always collided with your bedtime in Iran, or even worse, with your favorite series, you still picked up with patience, love, and just enough encouragement to keep me going.

To my distinguished siblings, Alireza, Shamimeh, and Amir, whose advice somehow managed to be more present in Canada than they ever were. As the youngest in the family, I have always had big footsteps to follow, and I am grateful that those footsteps were marked by laughter, example, and just the right amount of challenge.

To my dear friends who became my hotline support team: Beril, my wonderful neighbor through Ottawa's snowstorms and heatwaves; Mahsa, my middle school sitmate who became my Canadian anchor, along with her mother, who not only treated me like her own daughter but also fed me like one; and to my beloved friends Nazi, Melika, Farzane, and Sahar, whose golden memories from Iran carried me through more than a few existential crises.

To my Ottawa communities, OCC and CSGSA, who provided not only laughter, debates, and winter escapes, but also a sense of belonging exactly when it was needed most.

And finally, to my younger and future self: to the determined girl who never let obstacles overshadow her dreams, and to the woman I hope will always remember that resilience and persistence can turn even the hardest paths into shining pearls. Here is to both of you for keeping me moving forward, and hopefully never letting me do another PhD.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the increasing complexity and scope of software systems, their dependability has become a critical concern in software engineering [2, 3]. As software becomes more integrated into everyday tasks, the impact of its failures increases exponentially. To mitigate these risks, both proactive failure prediction and reactive anomaly detection techniques have been widely explored [4, 5].

Logs provide a source of information about system execution, capturing runtime behaviors that can be used to anticipate failures and identify anomalous patterns. Log-based failure prediction focuses on identifying patterns in log data that indicate impending failures. Due to the difficulty in benchmarking datasets on this task and security concerns about publishing labeled data, there have been limited studies on automated log-based failure predictors. The main studies leveraged DL techniques, particularly RNNs and transformers, which have shown effective performance [6, 7]. Log-based anomaly detection, on the other hand, has emerged as a reliable technique for ensuring system dependability. Unlike failure prediction, which aims to predict system failures proactively, anomaly detection focuses on identifying deviations from normal system behavior. Due to the availability of public datasets for anomaly detection, numerous studies have proposed various ML-based methods [8–11].

As mentioned above, despite the importance of failure prediction tasks, due to limited studies, we identified two main challenges in the literature: (1) lack of studies evaluating the impact of different DL network choices, combined with various embedding strategies, on log-based failure prediction, and (2) lack of various labeled datasets with different characteristics, such as dataset size and percentage of failure, to derive generalizable guidelines for future practitioners.

Regarding log-based anomaly detection, several challenges remain as well: (1) ML-based anomaly detection, particularly DL models, requires a substantial amount of labeled data, which is costly to obtain; (2) most available datasets suffer from data leakage due to overlap between training and test sets, inflating the reported effectiveness of super-

vised methods [12]; and (3) existing approaches assume stable log distributions, which is unrealistic as logs evolve due to software updates and environmental changes [13, 14].

Thus, improving failure prediction and anomaly detection techniques is crucial to ensure the reliability of modern software systems. While DL models have significantly improved log-based analysis, challenges related to the availability of labeled data, dataset generalizability, and log evolution must still be addressed. By tackling these challenges, research can contribute to the development of more reliable and scalable approaches, ultimately improving the dependability of software-intensive systems [6, 7, 11, 14–16]. Moreover, enhancing these techniques is required for dependable software evolution, where frequent updates and deployments in CI/CD pipelines evolve system behavior. Robust and efficient log analysis facilitates early fault detection and stable releases, strengthening the reliability and automation goals of modern DevOps practices.

1.2 Thesis Objectives

The long-term vision of this research is to provide accurate and generalizable methods for log analysis with ML components, including LLMs. More specifically, we describe in detail our two objectives:

TO_1 (Evaluation of DL-based Failure Predictors): The objective of this part is to systematically investigate the application of four main DL encoders on log-based failure prediction. The goal is to develop practical guidelines for utilizing DL-based failure prediction models, taking into account dataset characteristics such as dataset size and failure rates.

TO_2 (Robust and Data-efficient Anomaly Detection): The objective of this section is to design an anomaly detector that achieves SOTA effectiveness on unstable data while requiring less labeled data for training/fine-tuning.

1.3 Contributions

This thesis addresses critical challenges in log-based failure prediction and anomaly detection through two novel approaches that improve effectiveness and data efficiency. The research makes four types of key advancements over existing approaches: (1) comprehensive methodology development, (2) extensive empirical validation, (3) practical guidelines for deployment, and (4) creation of reproducible and reusable benchmarks. The specific contributions are organized as follows:

TO_1: Designing and evaluating of DL-based Failure Predictors. This contribution systematically investigates how different DL encoders and embedding strategies can be applied to log-based failure prediction, to provide actionable guidance to practitioners. The main contributions are:

- **Large-scale Systematic Evaluation:** We performed a comprehensive evaluation of four major DL encoders (LSTM, BiLSTM, CNN, and Transformer-based) and three embedding strategies (Logkey2vec, BERT, and a hybrid FastText+TF-IDF), systematically covering the main DL architectures used in log analysis.
- **Automated Dataset Synthesis:** We proposed a systematic and automated approach to generate synthetic log datasets, enabling controlled experiments by varying key characteristics such as dataset size, failure percentage, log sequence length, and failure pattern type. This allowed us to conduct fine-grained evaluations while avoiding various forms of bias.
- **Empirical Validation on Synthetic and Real-world Data:** We evaluated 360 synthetic datasets derived from three behavioral models, and compared results against real-world datasets. Results consistently confirm that CNN encoders with Logkey2vec achieve the best overall performance, especially when the dataset size exceeds 350 or the failure rate surpasses 7.5%.
- **Comparison with Traditional ML:** We benchmarked DL-based predictors against the best-performing traditional ML method (Random Forest), showing that DL configurations achieve significantly higher accuracy and robustness under realistic conditions.
- **Guidelines for Practice:** Based on extensive experiments, we provide practical guidelines for selecting DL-based failure predictors according to dataset characteristics, enabling practitioners to determine the best configurations of the DL-based failure predictor based on dataset characteristics.
- **Reproducibility and Benchmark Creation:** We release a publicly available replication package containing the implementation, generated datasets, and results, supporting reproducibility and enabling meaningful comparison for future research [17].

TO_2: Robust and Data-efficient Anomaly Detection. This contribution proposes a hybrid anomaly detection approach tailored for unstable logs (ULAD), where system or environment changes impact log structures. The key contributions include:

- **Configuration of Unstable Log Datasets:** We configure four unstable log datasets (ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U), ensuring realistic evaluation by introducing disparities between training and testing sets and eliminating potential data leakage.
- **Design of FlexLog:** We developed FlexLog, a novel hybrid approach that combines a fine-tuned LLM (Mistral) with simple ML models (KNN, DT, and SLFN) using ensemble learning. To improve practicality, FlexLog integrates Parameter-Efficient Fine-Tuning (PEFT), Retrieval-Augmented Generation (RAG), and a cache mechanism for efficient inference.

- **State-of-the-art Effectiveness and Data Efficiency:** We conducted extensive experiments on two real-world and two synthetic unstable log datasets. Results show that FlexLog consistently outperforms state-of-the-art baselines by at least 1.2 pp in F1 score, while reducing labeled data requirements by over 62.87 pp. On ADFA-U, FlexLog achieves up to a 13 pp increase in F1 score when trained on only 500 samples.
- **Efficiency Analysis:** We showed that FlexLog achieves these improvements while maintaining inference time under one second per log sequence and cache memory usage within practical limits, making it suitable for most production environments.
- **Reproducibility:** We provide all datasets, implementations, and replication packages to facilitate future research and ensure reproducibility [18].

By addressing TO_1 and TO_2, this thesis contributes both a systematic empirical foundation for DL-based log failure prediction and a practical, robust method for anomaly detection on unstable logs. Together, these contributions advance the state of the art for accurate and practical ML- and LLM-based log analysis, while supporting reproducibility and future studies in the field.

1.4 Publications

The research presented in this thesis has led to the following peer-reviewed publications: one in *Empirical Software Engineering (EMSE, Springer)* and the other in *ACM Transactions on Software Engineering and Methodology (TOSEM)*:

1. Fatemeh Hadadi, Joshua H. Dawes, Donghwan Shin, Domenico Bianculli, and Lionel Briand. "Systematic Evaluation of Deep Learning Models for Log-based Failure Prediction" in Empirical Software Engineering vol. 29, no. 105, June 2024.

   Presentation: This work was presented at the Journal First Track at ISSRE 2024 in Tsukuba, Japan.

   DOI: `https://doi.org/10.1007/s10664-024-10501-4`

2. Fatemeh Hadadi, Qinghua Xu, Domenico Bianculli, and Lionel Briand. "LLM meets ML: Data-efficient Anomaly Detection on Unstable Logs" in ACM Transactions on Software Engineering and Methodology 2025.

   Presentation: This work is submitted to be presented at the Journal First Track at ICSE 2026 in Rio de Janeiro, Brazil.
   DOI: `https://doi.org/10.1145/3771283`

1.5 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 addresses TO_1 by providing:
  - Background on the definition of failure prediction and its difference with other log-analysis tasks, such as anomaly detection, DL encoders, and embedding strategies for failure prediction,
  - A review of the related work and comparison with our systematic evaluation,
  - A modular architecture to assess various combinations of DL encoders and embedding strategies in a controlled way,
  - A through empirical evaluation of different DL-based failure predictors, along with a description of our algorithm to generate various datasets in a controlled way to derive comprehensive guidelines,
  - A discussion of the practical implications and potential threats to the validity of the study,
  - A conclusion for our work and suggestions for future work.
- Chapter 3 addresses TO_2 by providing:
  - Background on unstable logs, anomaly detection on unstable logs, and task adaptation strategies for LLMs,
  - A robust and efficient anomaly detection approach using ensemble learning of simple ML models with a fine-tuned LLM,
  - A demonstration of the effectiveness and efficiency of our proposed method compared to baselines,
  - A discussion on the token consumption of different LLM choices and potential threats to the validity of the study,
  - Review of the related works, including the application of LLMs for log analysis,
  - A conclusion for our work and possible future work.
- Finally, Chapter 4 presents a summary of the thesis contributions, along with discussions and suggestions for future work.

Chapter 2

Systematic Evaluation of Deep Learning Models for Log-based Failure Prediction

This chapter addresses RQ_1, i.e., systematic evaluation of DL models for log-based failure prediction models. The contents of this chapter have been published in the Journal of *Empirical Software Engineering (EMSE)* [17].

2.1 Overview

As discussed in Chapter 1, as software systems continue to increase in complexity and scope, reliability and availability play a critical role in quality assurance and software maintenance [2, 3]. During runtime, software systems often record log data about their execution, designed to help engineers monitor the system's behaviour [4]. One important quality assurance activity is to predict failures at run time based on log analysis, as early as possible before they occur, to enable corrective actions and minimise the risk of system disruptions [5].

However, software systems typically generate a vast quantity of log data, which makes manual analysis error-prone and extremely time-consuming. Therefore, a number of automatic log analysis methods, particularly for failure prediction [6, 7, 15] and anomaly detection [11, 14, 16], have been proposed over the past few years. Machine Learning (ML) has played a key role in automatic log analysis, ranging from traditional ML methods (e.g., Random Forest (RF) [19], Support Vector Machine (SVM) [20], Gradient Boosting (GB) [21]) to Deep Learning (DL) methods (e.g., DeepLog [11], LogRobust [14], LogBERT [10]). Unlike traditional ML models that often rely on manually designed features, such as log frequency counts, DL models can directly process higher-dimensional representations of log data in the form of embedding vectors, while also capturing contextual and sequential dependencies through their powerful architectures (e.g., Long Short-Term Memory (LSTM), Convolutional Neural Networks (CNN), and transformers [3]).

Although several studies have explored the use of DL models with various log sequence embedding strategies [4], they have been limited in terms of evaluating the three main types of DL networks—RNN, CNN, and transformer—combined with different embedding strategies; for instance, two studies by Le [3] and Lu [22] included CNN-based models but did not cover transformer-based models. Moreover, previously studied models were often applied to a limited number of available datasets, which severely limited the generalizability of results [4]. Indeed, because these few datasets exhibit a limited variety of characteristics, studying the robustness and generalizability of DL models, along with their embedding strategies, is unlikely to yield practical guidelines.

In this chapter, we aim to systematically investigate the combination of the main DL architectures and embedding strategies, based on datasets whose main characteristics (e.g., dataset size and failure percentage) are controlled. To achieve this, we first introduce a modular architecture for failure prediction, where alternative log embedding strategies and DL models can be easily applied. The architecture consists of two major steps: an embedding step that converts input logs into log embedding vectors followed by a classification step that predicts failures by processing the embedding vectors using encoders that are configured by different DL models, called DL encoders. In the embedding step, three alternative strategies, i.e., a semantic-based strategy (BERT [23]), a template ID-based strategy Logkey2vec [22], and aggregation of semantic and template ID-based strategies, FastText with TF-IDF [14], are considered. In the classification step, we apply four types of DL models—LSTM [24], BiLSTM [25], CNN [26], and Transformer [27]—to process the log embeddings.

Furthermore, we compared the results of our systematic investigation of DL architectures with a top traditional ML-based failure predictor to assess the advantage of DL-based approaches.

Also, to address the issue of the limited availability of adequate datasets, we designed a rigorous approach for generating synthesised data relying on behavioural models built by applying model inference algorithms [28, 29] to available system logs. When synthesising data, we control key dataset characteristics such as the size of the dataset and the percentage of failures. Additionally, we define patterns that are associated with system failures and are used to classify logs for the failure prediction task. The goal is to associate failures with complex patterns that are challenging for failure prediction models. Further, based on our study, we investigated how the dataset characteristics determine the accuracy of model predictions and then derive practical guidelines.

Finally, we processed a real-world dataset for failure prediction, called OpenStack_FP, to compare the results obtained on synthesised data with those obtained on a real-world failure prediction dataset. The objective was to obtain further evidence of the validity of our data synthesis strategy.

Our empirical results indicate that the best model incorporates a CNN-based encoder with Logkey2vec as the embedding strategy. Using a wide variety of datasets, both synthesised and real-world, showed that this combination is also very accurate when certain conditions are met, specifically in terms of dataset size and failure percentage. Our findings offer valuable insights for software and AIOps engineers to select the most suitable

DL-based solution for optimal failure prediction. Moreover, we aim to provide guidance for optimizing dataset characteristics to improve failure prediction accuracy. In conclusion, this chapter offers clear guidelines for those looking to leverage DL in predicting system failures from logs.

To summarise, the main contributions of this chapter are:

- A large-scale, systematic investigation of the application of various DL encoders—LSTM-, BiLSTM-, CNN-, and transformer-based—and embedding strategies—BERT [23], Logkey2vec [22] and hybrid strategy combining FastText with TF-IDF [14]—for failure prediction modeling
- A systematic and automated approach to synthesise log data, with a focus on experimentation in the area of failure prediction, to enable the control of key data set characteristics while avoiding any other form of bias.
- A comparison of the results obtained on synthesised data with those of a real-world dataset to provide further evidence of the validity of our data synthesis strategy.
- A comparison of DL-based and one of the best-performing traditional ML-based failure predictors to assess the benefits of the former.
- Practical guidelines for using DL-based failure prediction models according to dataset characteristics such as dataset size and failure rates.
- A publicly available replication package, containing the implementation, generated datasets with behavioural models, and results.

The rest of the chapter is organised as follows. Section 2.2 presents the basic definitions and concepts that will be used throughout the chapter. Section 2.3 illustrates related work. Section 2.4 describes the architecture of our failure predictor, including its various configuration options. Section 2.5 describes our research questions, empirical methodology, and synthetic log data generation. Section 2.6 reports empirical results. Section 2.7 discusses the implications of the results. Section 2.8 concludes the chapter and suggests future directions for research and improvements.

2.2 Background

In this section, we provide background information on the main concepts and techniques that will be used throughout the chapter. First, we briefly introduce the concepts related to Finite State Automata (FSA) and regular expressions in § 2.2.1 and execution logs in § 2.2.2. We then describe two important log analysis tasks (anomaly detection and failure prediction) in § 2.2.3 and further review ML-based approaches for performing such tasks in § 2.2.4. We conclude by providing an overview of embedding strategies for log-based analyses in § 2.2.5.

2.2.1 Finite State Automata and Regular Expressions

A *deterministic FSA* is a tuple $\mathcal{M} = \langle Q, A, q_0, \Sigma, \delta \rangle$, where Q is a finite set of states, $A \subseteq Q$ is the set of accepting states, $q_0 \in Q$ is the starting state, Σ is the alphabet of the automaton, and $\delta : Q \times \Sigma \rightarrow Q$ is the transition function. The extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$, where Σ^* is the set of strings over Σ, is defined as follows:

(1) For every $q \in Q, \delta^*(q, \epsilon) = q$, where ϵ represents the empty string;

(2) For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$, $\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$.

Let $x \in \Sigma^*$; the string x is accepted by $\mathcal{M}$ if $\delta^*(q_0, x) \in A$ and is rejected by $\mathcal{M}$, otherwise.

The language accepted by an FSA $\mathcal{M}$ is denoted by $\mathcal{L}(\mathcal{M})$ and is defined as the set of strings that are accepted by $\mathcal{M}$; more formally, $\mathcal{L}(\mathcal{M}) = \{w \mid \delta^*(q_0, w) \in A\}$. A language accepted by an FSA is called a *regular* language.

Regular languages can also be defined using *regular expressions*; given a regular expression r we denote by $\mathcal{L}(r)$ the language it represents. A regular expression r over an alphabet Σ is a string containing symbols from Σ and special meta-symbols like "|" (union or alternation), "." (concatenation), and "*" (Kleene closure or star), defined recursively using the following rules:

(1) $\emptyset$ is a regular expression denoting the empty language $\mathcal{L}(\emptyset) = \emptyset$;

(2) For every $a \in \Sigma$, a is a regular expression corresponding to the language $\mathcal{L}(a) = \{a\}$;

(3) If s and t are regular expressions, then $r = s|t$ and $r = s.t$ (or $r = st$) are regular expressions denoting, respectively, the union and the concatenation of $\mathcal{L}(s)$ and $\mathcal{L}(t)$;

(4) If s is a regular expression, then $r = s^*$ is a regular expression denoting the Kleene closure of $\mathcal{L}(s)$.

2.2.2 Logs

In general, a *log* is a sequence of log messages generated by logging statements (e.g., `printf()`, `logger.info()`) in the source code [4]. A *log message* is textual data composed of a *header* and *content* [4]. In practice, the logging framework determines the *header* (e.g., `INFO`) while the *content* is designed by developers and is composed of static and dynamic parts. The static parts are the fixed text written by the developers in the logging statement (e.g., to describe a system event), while the dynamic parts are determined by expressions (involving program variables) evaluated at runtime. For instance, let us consider the execution of the log printing statement `logger.info(``Received block␣"+ block_ID)`; during the execution, assuming variable `block_ID` is equal to 2, the log message `Received block 2` is printed. In this case, `Received block␣` is the static part while 2 is the dynamic part, which changes depending on the value of `block_ID` at run time.

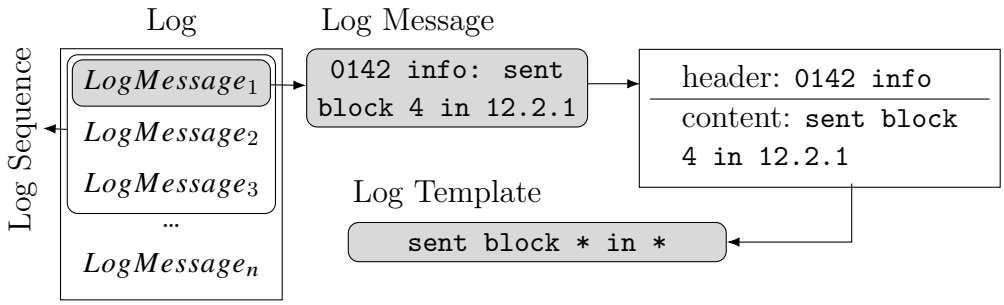


Figure 2.1: An example illustrating the concepts of log, log message, log template, and log sequence

A *log template* (also called *event template* or *log key*) is an abstraction of the log message content, in which dynamic parts are masked with a special symbol (e.g., `*`); for example, the log template corresponding to the above log message is `Received block␣*`. Often, each unique log template is identified by an ID number for faster analysis and efficient data storage.

A *log sequence* is a fragment of a log, i.e., a sequence of log messages contained in a log; in some cases, it is convenient to abstract log sequences by replacing the log messages with their log templates. Log sequences are obtained by partitioning logs based on either log message identifiers (e.g., session IDs) or log timestamps (e.g., by extracting consecutive log messages using a fixed/sliding window). For a log sequence l, $|l|$ indicates the length of the log sequence, i.e., the number of elements (either log templates or log messages), not necessarily unique, inside the sequence.

Figure 2.1 shows an example summarizing the aforementioned concepts. On the left side, the first three log messages are partitioned (using a fixed window of size three) to create a log sequence. The first message in the log sequence (*LogMessage1*) is `0142 info: sent block 4 in 12.2.1`. It is decomposed into the header `0142 info` and the content `sent block 4 in 12.2.1`. The log template for the content is `sent block * in *`; the dynamic parts are `4` and `12.2.1`.

2.2.3 Log Analysis Tasks

In the area of log analysis, several major tasks for reliability engineering, such as anomaly detection, and failure prediction, have been automated [4]; we provide an overview of these tasks below.

2.2.3.1 Anomaly Detection

Anomaly detection is the task of identifying anomalous patterns in log data that do not conform to expected system behaviours [4], indicating possible errors, faults, or failures in software systems.

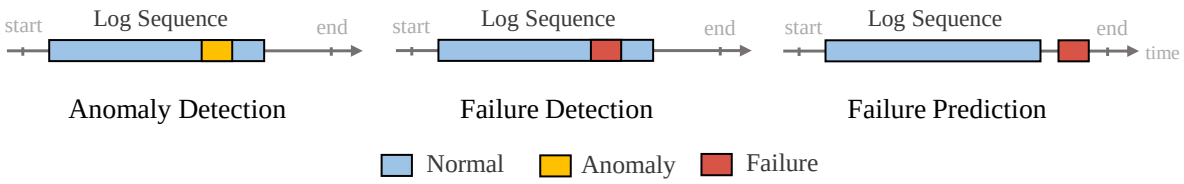


Figure 2.2: Illustration of Log Analysis Tasks.

To automate the task of anomaly detection, log data is often partitioned into smaller log sequences. This partitioning is typically based on log identifiers (e.g., *session_ID* or *block_ID*), which correlate log messages within a series of operations; alternatively, when log identifiers are not available, timestamp-based fixed/sliding windows are also used. Le [3] assessed the accuracy of anomaly detection models considering both timestamp-based partitioning (with different time periods) and log identifier partitioning; models achieved higher accuracy and exhibited robustness when using the latter.

Labelling of partitions is then required, with each partition typically labelled as an anomaly when an error, unknown, or failure message appears within it, or when the corresponding log identifier is marked as anomalous. Otherwise, it is labelled as normal.

Failure Detection. Failure detection is a special type of anomaly detection that specifically identifies failures within logs [30], as compared in Figure 2.2. Similar to anomaly detection, log data is partitioned into sequences. The decision of whether a log should be tagged as anomalous or a failure depends on the system being analysed. By definition, anomaly detection targets a wide scope of abnormal behaviours (which may or may not be a system failure) whereas failure detection focuses on system failures.

2.2.3.2 Failure Prediction

Failure prediction attempts to proactively generate alerts *before* the occurrence of failures, which often lead to unrecoverable outages [4]. In failure prediction, a log is partitioned similarly to previous tasks, often using a session-based log identifier.

The main differences between failure prediction and the above tasks are the following:

- *mode of operation.* As shown in Figure 2.2, anomaly and failure detection are reactive approaches that raise a flag once an anomaly or failure has happened. Instead, failure prediction is *proactive*. It forecasts potential future failures, allowing enough time to address them.
- *input data.* The input for failure prediction typically consists of normal-looking inputs, a subset of which contains subtle, complex patterns in logs that may be associated with future failures. Patterns can indicate impending issues that have not yet manifested as failures in log data.

Figure 2.3 shows a simplified comparison of "positive sequences" (in contrast to "normal" sequences) for the aforementioned tasks (depicted in Subfigures 2.3b, 2.3c, and 2.3d),

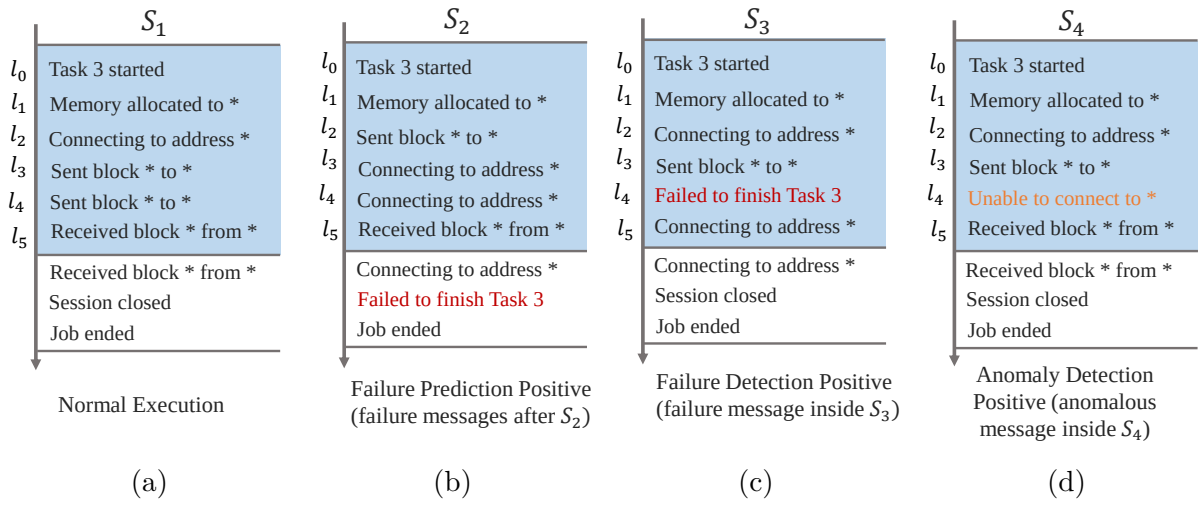


Figure 2.3: Comparison of Normal Sequence (on the left) and Positive Sequences in Log Analysis Tasks (on the right).

next to a normal log (depicted in Subfigure 2.3a). The blue box in each Subfigure highlights a partitioned sequence of log templates, labelled as S_1, S_2, S_3, and S_4. For failure prediction (see Subfigure 2.3b), log templates in S_2 look normal when considered individually. However, their occurrence creates a pattern indicating a point on the timeline where a future failure, highlighted in red, happens. Hence, S_2 is a positive case in data labelling for failure prediction. Subfigure 2.3c, on the other hand, shows S_3 as a positive instance for failure detection, since there is a failure message (also highlighted in red) within the blue sequence. Similarly, in anomaly detection, an anomalous log message, highlighted in yellow, appears within S_4 (see Subfigure 2.3d).

Dataset Transferability for Failure Prediction. It is worth mentioning that, as sketched in Subfigure 2.3d, one cannot necessarily expect the occurrence of a failure after a log sequence with an anomalous section. That is, log data used for anomaly detection are not interchangeable with those intended for failure prediction. Therefore, using anomaly detection data for failure prediction would likely yield inaccurate and misleading results.

When using data specifically intended for failure detection in the context of failure prediction, certain assumptions should be met. First, log data must be ordered by timestamp to enable the separation of message sequences before a failure occurs. In addition, one must rigorously label log data to decide whether a sequence of log messages is indeed related to a future failure; this is especially challenging when there is no clear evidence of a failure, unlike many anomaly detection datasets. The quality of the initial labelling plays an important role as well. Considering the above, log data labelled for failure detection can be used for predictive tasks through careful preprocessing and rigorous validation (see § 2.5.2.3).

2.2.4 DL Techniques in Log Analysis

In recent years, a variety of DL techniques have been applied to log analysis, and more specifically to failure prediction and anomaly detection. Compared to traditional ML techniques such as Random Forests (RF) and K-nearest Neighbours (KNN), DL techniques incrementally learn high-level features from data, removing complex feature extraction activities based on domain expertise.

According to Le [3], there are three main categories of DL approaches in log analysis: (1) Recurrent Neural Network (RNN), (2) Convolutional Neural Network (CNN), and (3) transformer. Additionally, we have a new growing category called (4) Graph Neural Network (GNN). In each category, different variations can be adopted; for instance, Long Short-Term Memory networks (LSTM) and Bidirectional Long Short-Term Memory networks (BiLSTM), which fall into the RNN category, have been repeatedly used for anomaly detection and failure prediction [6, 11, 14]. We now explain the major features of each category as well as their variations.

2.2.4.1 RNN

LSTM [31, 32] is an RNN-based model commonly used in both anomaly detection and failure prediction [6, 11]. An LSTM network consists of multiple units, each of which is composed of a cell, an input gate, an output gate [31], and a forget gate [32]. An LSTM-based model reads an input sequence $(x_1, \ldots, x_n)$ and produces a corresponding sequence $(y_1, \ldots, y_n)$ with the same length. At each time step $t > 1$, an LSTM unit reads the input x_t as well as the previous hidden state h_{t-1} and the previous memory c_{t-1} to compute the hidden state h_t. The hidden state is employed to produce an output at each step. The memory cell c_t is updated at each time step t by partially forgetting old, irrelevant information and accepting new input information. The forget gate f_t is employed to control the amount of information to be removed from the previous context (i.e., c_{t-1}) in the memory cell c_t.

As a recurrent network, an LSTM shares the same parameters across all steps, which reduces the total number of parameters to learn. Learning is achieved by minimizing the error between the actual output and the predicted output. Moreover, to improve the regularization of an LSTM-based model, a dropout layer is applied between LSTM layers. It randomly drops some connections between memory cells by masking their value. LSTM-based models have shown significant performance in several studies in log-based failure prediction and anomaly detection [6, 7, 16, 33].

BiLSTM is an extension of the traditional LSTM [34]. However, BiLSTM reads the sequence in both directions, enabling it to comprehend the relationships between the previous and the upcoming inputs. To make this possible, a BiLSTM network is composed of two layers of LSTM nodes, whereby each of these layers learns from the input sequence in the opposite direction. At time step t, the output h_t is calculated by concatenating h_t^f (the hidden states in a forward pass) and h_t^b (the hidden states in a backward pass). By allowing this bi-directional computation, BiLSTM can capture complex dependencies

and produce more accurate predictions. The BiLSTM-based model has achieved accurate results for anomaly detection [14].

2.2.4.2 CNN

CNN is a neural network primarily employed for image recognition [26]. It has a unique architecture designed to handle 2D and 3D input data such as images and matrices. A CNN leverages convolutional layers to perform feature extraction and pooling layers to downsample the input.

The 1D convolutional layer uses a set of filters to perform convolution operation with the 2D input data to produce a set of feature maps (CNN layer output). According to Kim [35], let $w \in R^{k \times d}$ be a filter which is applied to a window of k elements in a d-dimension input log sequence, and let x_i represent the i-th elements in the sequence. A feature $c_i \in R$ is calculated as $c_i = \sigma(w.x_{i:i+k-1} + b)$, where σ is the activation function (i.e., $ReLu$), $x_{i:i+k-1}$ represents the concatenation of elements $\{x_i, x_{i+1}, ..., x_{i+k-1}\}$, and $b \in R$ denotes a bias term. After this filter is applied to each window in the sequence ($\{x_{1:k}, x_{2:k}, ..., x_{n-k+1:n}\}$), a feature map $c = [c_1, c_3, ..., c_{n-k+1}]$ is produced, where $c \in R^{n-k+1}$. Parameter k represents the kernel size; it is as an important parameter of the operation. Note that no padding is added to the input sequence, resulting in feature maps that are smaller than the input sequence. Padding is a technique used to add zeros to the beginning and/or end of the sequence, allowing for more space for the filter to cover and controlling the size of the output feature maps. Padding is commonly used so that the output feature map has the same length as the input sequence [36].

The pooling layer reduces the spatial dimensions of the feature maps extracted by the convolutional layer, thereby simplifying the computational complexity of the network.

Recently, CNNs have shown high-accuracy performance in anomaly detection [22].

2.2.4.3 Transformer

The transformer is a type of neural network architecture designed for natural language processing tasks, introduced by Vaswani [27]. The main innovation of transformers is the self-attention mechanism. More important parts of the input receive higher attention, which facilitates learning the contextual relationships from input data. This is implemented by calculating a weight for each input element, which represents the importance of that element relative to its adjacent elements. Hence, a model with self-attention (not necessarily a transformer) can capture long-range dependencies in the input. Since the transformers do not process inputs sequentially like LSTM, positional encoding is needed. Positional encoding vectors are fixed-size, added to the input to provide information about the position of each element in the input sequence. Further, a transformer involves a stack of multiple transformer blocks. Each block contains a self-attention layer and a feed-forward neural network layer. In the self-attention layer, the model computes attention scores (weights) for each element, allowing it to capture the relationship between all input elements. The feed-forward layer is used to transform the representation learned by the

self-attention layer into a new representation entering the next transformer block. In the area of log analysis, transformers have been recently applied in a few studies on anomaly detection [10, 37–39], showing outstanding performance.

2.2.4.4 GNN

A Graph Neural Network (GNN) is a neural network designed to process data structured as graphs [40]. During training, it takes a graph-structured input and updates node feature vectors (where nodes are equivalent to vertices in the graph) iteratively with respect to the feature vectors of its neighbor nodes and itself. By using the final feature vectors, GNNs can discern intricate relationships within the graph data. Hence, GNNs can be used for classification tasks at either the graph or node level.

The main difference between GNN and the aforementioned DL techniques lies in the data structure they process. GNNs process data structured as graphs. Since log data is initially sequential, it requires further processing to construct a graph from it. When a node represents a log template and a graph corresponds to a log sequence, classification at the graph level requires an aggregation method such as READOUT [40] to combine node feature vectors.

We note that GNNs are sometimes regarded as a representation method, such as log sequence embedding strategies detailed in § 2.2.5, since they compute the graph representation of log sequences [41]. However, we consider them as a classification method, like the other DL methods described in § 2.2.4, since current GNNs necessitate pre-existing semantic embeddings for input.

2.2.5 Log Sequence Embedding Strategies

When analyzing log sequences, the textual data of log sequence elements must be converted into a vector representation that is understandable by a machine; this conversion is called the *log sequence embedding*. Generally, there are three main approaches for doing this: (1) template ID-based strategies such as count vectors [42], (2) semantic-based strategies based on the contextual information of sequence elements, or (3) hybrid strategy as a combination of the previous two strategies. Here, we cover one widely used example for each case in the following sections.

2.2.5.1 Template ID-based Strategy

Many studies have achieved high accuracy results by using log embedding strategies that rely on the ID numbers or count vectors of log sequence elements [43]. Advantages include the speed of processing and model simplicity since text preprocessing (e.g., tokenization) is not required. However, they do not consider the order of log messages (templates) in a log sequence, making them prone to unreliable results when the sequential pattern of log messages (templates) matters (e.g., in failure prediction).

TF-IDF [44] is a widely used embedding strategy in data mining and information retrieval, employed for log analysis at two different levels: log template ID level and word (token) level. At the log template ID level, it measures the frequency of a unique log template in a log sequence, Term Frequency (TF), divided by how common this log template is in the total dataset (i.e., Inverse Document Frequency - IDF). At the word (token) level, it delves deeper. It calculates the TF-IDF value for each unique word (token) inside a log template and assigns the aggregated value to a log template. Both TF-IDFs compute an embedding vector for each log sequence, making them incompatible with methods that require an embedding vector at the log template level.

Logkey2vec, introduced by Lu [22], is another strategy used in log analysis, which is based on log template IDs and enables the transformation of a log template into an embedding vector. Logkey2vec maps each unique log template ID to a vector representation. It is a trainable layer implemented inside a neural network. It relies on a matrix called "codebook", where the number of rows is the vocabulary size and the number of columns is the embedding vector size of each log template ID. The embedding vectors are first initialised with random numbers and are refined through backpropagation during training. For a log sequence, Logkey2vec computes the embedding vector of each log template based on its log template ID; each row of the matrix represents the whole log sequence. We note that Logkey2vec is not semantic-based in a linguistic sense since it solely takes log template IDs as input, disregarding the semantic information that lies in the text of log templates. Moreover, unlike tools such as word2vec [45], which is pre-trained using CBOW (Continuous Bag-of-Words) and Skip-grams [45], Logkey2vec is not pre-trained by any method; it requires the aforementioned training on its target log data. This strategy has also been applied, with a different name, by Bogatinovski [30] (who used the term "vectorizer"), and by [10] (who used the term "Embedding Matrix").

2.2.5.2 Semantic-based Strategy

Studies using semantic-based strategies take into account the linguistic relationship between words in log templates. In 2019, Meng [16] proposed *template2vec*, an embedding strategy based on the synonyms and antonyms relation of words mentioned in log data. This strategy enables the matching of new log templates with existing ones. However, since it is trained on manually added domain-specific synonyms and antonyms, its applicability is limited.

In the past few years, Bidirectional Encoder Representations from Transformers (BERT) has provided significant improvements in the semantic embedding of textual information by taking into account the contextual information of text. It has been used in a few studies in log sequence embedding [10, 37]. This model outperforms the other pretrained transformer-based models, GPT2 [46] and RoBERTa [47], in log sequence embedding [37].

The pre-trained BERT base model [23] provides the embedding matrix of log sequences, where each row is the representation vector of its corresponding log template inside the sequence. The BERT model is applied to each log template separately, and then the representation is aggregated into a matrix. To embed the information of a log template

into a 768-sized vector, the BERT model first tokenizes the log template text. BERT tokenizer uses WordPiece [48], which can handle Out-Of-vocabulary (OOV) words to reduce the vocabulary size. Further, the tokens are fed to the 12 layers of BERT's transformer encoder. After obtaining the output vectors of a log template's tokens, the log template embedding is calculated by getting the average of the output vectors. This process is repeated for all the log templates inside the log sequence to create an $n \times 768$ matrix representation where n is the size of the log sequence.

2.2.5.3 Hybrid Strategy

This category aims to combine the benefits of both template ID-based and semantic-based strategies, compensating for their respective limitations. The main instance of this category is the study by Zhang [14]. They leverage FastText [49] to convert each word (or token) of a log template into a d-dimensional vector ($d = 300$). FastText is a word vectorisation tool pre-trained on the Common Crawl Corpus dataset [50]; it converts words into vectors while capturing their semantic relationship. Consequently, words having similar meanings result in similar vectors. The word vectors are further aggregated into a single vector representing a log template, calculated using a weighted average with TF-IDF (computed at the word level). Specifically, consider a log template T consisting of a list of words, $[t_1, t_2, ..., t_N]$, where N indicates the number of words. The list of words can be represented as a list of vectors $[v_1, v_2, ..., v_N]$, where $v_i \in R^d$ is a semantic vector of t_i. The embedding vector of T, V_T, is then calculated according to Equation 2.1, where $w_i \in R$ indicates the TF-IDF value of t_i.

$$V_T = \frac{1}{N} \sum_{i=1}^{N} w_i . v_i \tag{2.1}$$

This strategy seeks to retain the advantages of the previous strategies. If a word is frequently mentioned among log templates, it is assigned a lower TF-IDF weight during the aggregation of word vectors, thereby increasing the distinction between embedding vectors of log templates. Moreover, similar to BERT but less informative in terms of word context, FastText assigns vectors with high cosine similarity to two log templates that contain different words but are semantically related.

2.3 Related Work

In this section, we will first discuss empirical studies on log-based anomaly detection and move on to more closely related failure prediction studies. We will also discuss studies related to dataset synthesis at the end.

2.3.1 Related Empirical Studies

2.3.1.1 Log-based Anomaly Detection

As discussed in § 2.2.3, anomaly detection is a different task than failure prediction. However, since they are both binary classification tasks on log data, they can rely on similar DL architectures [6, 7]. Several papers report empirical studies of different DL-based methods for log-based anomaly detection. Due to the large number of works and differences in objectives, in our review, we include studies that covered more than one DL model, possibly based on the same DL-based approaches.

Table 2.1 briefly summarises anomaly detection studies, including empirical evaluations. Column "DL Type(s)" indicates the type of DL network covered in each paper. We indicate the Log Sequence Embedding (LSE) strategies, introduced in § 2.2.5, in the next column; notice there are a few models not using LSE, such as DeepLog [11]. Column "Dataset(s)" indicates which datasets (whether existing datasets or synthesised ones) were used in the studies. Column "Control of Dataset Characteristics" indicates whether the dataset characteristics were controlled during the experiment and lists such characteristics. In the last column, the labelling scheme indicates the applied method(s) for log partitioning, as mentioned in § 2.2.2, based either on a log identifier or on timestamp (represented by L and T, respectively).

We now briefly explain the included papers to motivate our study and highlight the differences. We note that, unless we mention it, LSE strategies are implemented specifically for one DL model (combinations are not explored). Indeed, many of the reported techniques tend to investigate one such embedding strategy or simply do not rely on any. The studies are listed in chronological order. Lu [22] (2018) introduced CNN for anomaly detection as well as the Logkey2vec embedding strategy (see § 2.2.5.1). They compared it to LSTM and MLP networks, also relying on the Logkey2vec embedding strategy. Meng [16] (2019) developed LogAnomaly, an LSTM-based model, using their proposed embedding strategy, Template2Vec (a log-specific variant of Word2Vec).

The first study considering transformers in their DL comparison is by Huang [38] (2020), featuring three DL models: HitAnomaly (transformer-based), LogRobust [14] (BiLSTM-based), and DeepLog (LSTM-based). HitAnomaly utilises transformer blocks (see § 2.2.4.3) as part of its LSE strategy, called Log Encoder. LogRobust employed the hybrid strategy of FastText and TF-IDF shows as F+T while DeepLog did not utilise any LSE strategy. The authors also controlled dataset characteristics by manipulating the unstable log ratios. Yang [51] (2021) proposed the GRU-based [54] PLELog and compared it to LogRobust and DeepLog. PLELog used the TF-IDF technique and LogRobust used F+T. Guo [10] (2021) proposed a transformer-based model, LogBERT, and compared its performance with two LSTM-based models, LogAnomaly and DeepLog. LogBERT uses an Embedding Matrix for its embedding strategy, which is similar to Logkey2vec. Le and Zhang [37] (2021) evaluated their proposed transformer-based model, Neurallog, against LogRobust (BiLSTM-base) and DeepLog (LSTM-based). The LSE strategies for the models were a pre-trained BERT (see § 2.2.5.2) for Neurallog and Log2Vec [33] (a strategy based on Word2Vec) for DeepLog.

Table 2.1: Overview of Related Empirical Studies

Paper	DL Type(s)	LSE Strategi(es)	Dataset(s)	Control of Dataset Characteristics	Labelling Scheme
Anomaly Detection					
Lu [22]	LSTM, CNN, MLP	Logkey2vec	HDFS	No	L
Meng et al. [16]	LSTM	template ID, Template2Vec	HDFS, BGL	No	T, L
Huang et al. [38]	LSTM, BiLSTM, Transformer	count vector, F+T, Log Encoder	HDFS, BGL, OpenStack	Yes (unstable log injection ratio)	T, L
Yang et al. [51]	LSTM, BiLSTM, GRU	template ID, TF-IDF, F+T	HDFS, BGL	No	T, L
Guo et al. [10]	LSTM, Transformer	template ID, count vector, Embedding Matrix	HDFS, BGL, Thunderbird	No	T, L
Le and Zhang 2021 [37]	LSTM, BiLSTM, Transformer	count vector, Log2Vec*, F+T, BERT	HDFS, BGL, Spirit, Thunderbird	No	T, L
Bogatinovski et al. [30]	LSTM, Transformer	count vector, vectorizer	OpenStack_v2	Yes (unstable log injection ratio)	T
Le and Zhang 2022 [3]	LSTM, BiLSTM, GRU, CNN	template ID, Logkey2vec, F+T	HDFS, BGL, Spirit, Thunderbird	Yes (class distribution, data noise, partitioning methods)	T, L
Xie [52]	BiLSTM, CNN Transformer, GNN	count vector, Logkey2vec, F+T, BERT	HDFS, BGL, Spirit, Thunderbird	Yes (partitioning methods)	T, L
Wu [41]	MLP, CNN LSTM	count vector, TF-IDF, Word2Vec, FastText, BERT	HDFS, BGL, Spirit, Thunderbird	Yes (partitioning methods)	T, L
Failure Prediction					
Lin [53]	BiLSTM	N/A	AzureML	No	T
Das et al. 2018 [6] 2020 [7]	LSTM	template ID	Clay-HPC	No	L
Our Study**	LSTM, BiLSTM, CNN, Transformer	Logkey2vec, BERT, F+T	synthesised Data, OpenStack_FP	Yes (Dataset size, Failure Percentage, LSL, Failure Pattern type)	L

**: we highlight that Log2Vec is different than Logkey2vec, a log sequence embedding strategy (see § 2.2.5.1) **: further discussed in § 2.7*

An important recent work on failure detection is the study of Bogatinovski [30]. They presented log data as sequences of subprocesses instead of sequences of log templates. To this end, they used transformer-based network and clustering methods to extract subprocesses and further leverage them to detect failure using an HMM [55]. For LSE, they designed the "vectorizer" that is similar to Logkey2vec. Their work includes the evaluation of varying unstable log ratios and their impact on their model performance.

Le [3] (2022) conducted a comprehensive evaluation of several DL models including LSTM-based models such as DeepLog and LogAnomaly, GRU-based model PLELog, BiLSTM-based model LogRobust, and CNN. The study focused on various aspects including data selection, data partitioning, class distribution, data noise, and early detection ability. Although they provide insights on many models and dataset characteristics, they did not include transformer-based models such as Neurallog, or recent semantic-based LSE strategies like BERT, and are limited to commonly used datasets.

Xie [52] (2022) proposed a GNN-based anomaly detection model, LogGD, and compared it with DL-based models from three categories: CNN, LogRobust (which is BiLSTM-based), and NeuralLog (which is transformer-based). Both NeuralLog and LogGD leverage BERT to extract semantic embeddings from log sequences. While Xie [52] took into account a wide range of DL techniques and LSE strategies from each category, their results, similar to those of Le [3] (2022), were obtained using only public datasets.

Finally, Wu [41] (2023) studied the effectiveness of different LSE on ML-based models for anomaly detection. In contrast to the study of Le [3], they explored all the possible combinations between LSE strategies and DL techniques and provided an accurate ranking for each category. They included six LSE strategies: Count Vector and TF-IDF (the word-level and template-level) as template ID-based strategies, and Word2Vec, FastText, and BERT as semantic-based strategies. However, they did not consider hybrid strategies. DL techniques are limited to MLP, CNN, and LSTM, while the rest of the common methods such as BiLSTM and transformers are left out. Similar to Le [3], their results are bound to four public datasets.

Datasets. Studies relying on publicly available datasets are limited to the following: Hadoop Distributed File System (HDFS) collected in 2009, and three HPC datasets, BGL, Spirit, and Thunderbird, collected between 2004 and 2006. Besides, for failure detection, there is the OpenStack dataset (2017) created by injecting a limited number of bugs at different execution points. In 2022, thanks to the effort of Bogatinovski [30], OpenStack was labelled at the log message level, which we refer to as OpenStack_v2. Overall, due to the limited number of available public datasets, there is a growing number of works focusing either on labelling existing data to a deeper level or on synthesising log data, as discussed in the following section.

2.3.1.2 Log-based Failure Prediction

In recent years, there have been a number of studies on log-based failure prediction, especially in large-scale systems where signs of failure may not be obvious. Early works on

failure prediction focused on structured logs (e.g., numeric parameters) mined from system logs. Sahoo [15] collected system health status logs and employed several time-series models such as the mean of previous values to predict indicative metrics (e.g., system utilization percentage, network IO usage, and system idle time). Russo [56] applied different SVMs relying on radial basis function and linear kernels that take multi-dimensional data representing values for each of the metrics to predict a future log sequence related to a failure. More recently, Lin [53] proposed a method that combines two ML models, BiLSTM and RF, to process temporal and spatial data, respectively, and concatenates their outputs to predict the likelihood of a node failing in the near future. Zhang [57] expanded this task to semi-structured logs. They extracted log templates from raw syslog messages and derived features from sequences of log templates. By training an RF-based model, Prefix, on features of previously seen log datasets, they achieved high accuracy in switch failure prediction compared to SVM and HMM. More recently, Liu [58] adopted machine learning models to predict system crashes on cloud service data; in their study, RF achieved the best accuracy compared to XGBoost and SVM. The study of Das [6] opened the door to analyzing semi-structured logs using DL. After extracting unique log templates, they derived patterns from them leading to a failure using LSTM. Following that, in 2020, they introduced an improved LSTM-based model, Aarohi [7], as state-of-the-art with faster inference time. Both Dash and Aarohi rely on the template ID-based strategy for embedding (see § 2.2.5). The above DL-based studies of failure prediction are briefly summarised in Table 2.1.

Datasets. Due to security concerns, in many reported works in the literature, the data sources are unavailable, including the Clay-HPC (Clay High-Performance Computing (HPC) systems) dataset applied on Aarohi [7] and Dash [6]. In contrast, the Prefix dataset is available but is of limited use; it was designed to maximise Mean Time Between False Alarms (MTBFA) and, due to its simple log patterns and anomaly cases, yields high accuracy regardless of the approach. As a result, we found the limited number of publicly available datasets to be a hindrance to our research. We therefore opted to develop a method for synthesising new datasets, as described next.

2.3.2 Dataset Synthesis Algorithms

In the log analysis literature, especially in anomaly detection, dataset synthesis refers to the modification of an existing dataset to simulate specific scenarios, such as system performance issues [59] or the evaluation of logs driven by system updates [14, 38]. On the other hand, in closely related literature on system monitoring, there are data synthesis algorithms for trace and benchmark generation that can create new data without relying on an existing dataset [60, 61]. Given the restrictions of available and suitable datasets for our failure prediction problem (as discussed in § 2.2.3), we henceforth refer to the second group of algorithms when mentioning data synthesis.

In 2005, Blom [60] proposed a method for generating test suites for systems whose behaviours can be described by Extended Finite State Machines (EFSM). This method

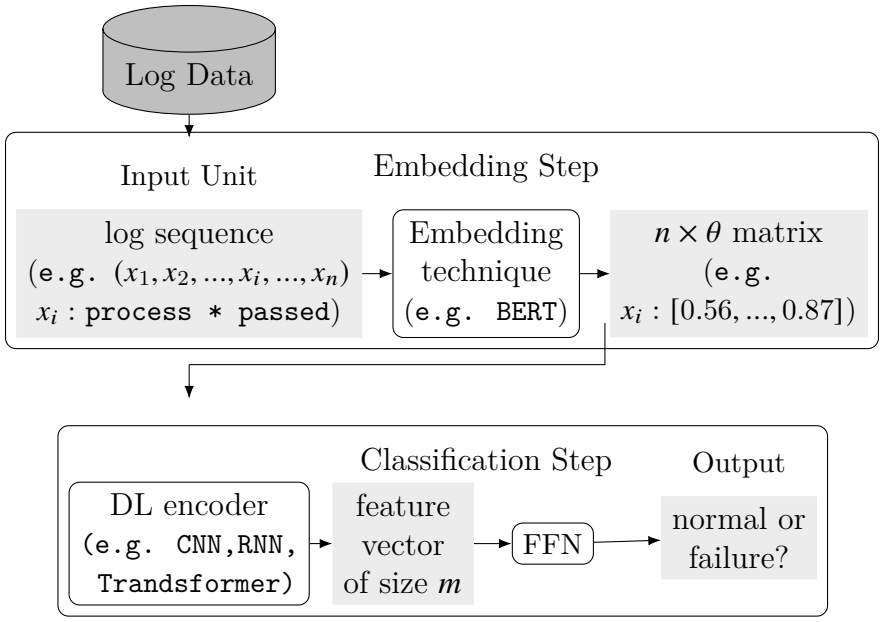


Figure 2.4: Overview of the modular architecture for failure prediction

produces a test sequence, referred to as a trace, that represents a coverage item. An *observer* monitors the trace and "accepts" it in case the specified coverage item has been covered. More recently, in 2017, Kluge [62] introduced EMSBench, which contains a model capable of mimicking complex system behaviour. Using this model, sequential traces are generated to compare different platforms. In 2020, Bombarda [61] leveraged FSM to design an algorithm that produces test sequences in the form of traces, identifying those with invalid inputs. By employing FSM, they successfully embedded the system constraints into the FSM during the generation process, ensuring the creation of only valid test sequences. Furthermore, Krstic [63] presented an algorithm for generating an event stream with their associated arbitrary values. These logs are compatible with system specifications in the Model for First-Order Dynamic Logic (MFODL) [64]. We will further discuss existing data synthesis algorithms and their differences with ours when we present the latter in § 2.5.3.

2.4 Failure Prediction Architecture

This section introduces our modular architecture for failure prediction, which aims to help us systematically evaluate various embedding strategies and DL encoders. Moreover, this modular architecture can serve as a baseline architecture for follow-up studies. Therefore, we describe it in this section, independently of the description of the empirical study design (see Section 2.5).

Figure 2.4 depicts the modular architecture. The architecture consists of two main steps, *embedding* and *classification*, allowing for different embeddings and DL techniques, respectively. We note that preprocessing is not required in this architecture since log sequences are based on log templates, which are already preprocessed from log messages.

In the embedding step, log sequences are given as input, and each log sequence is in the form $(x_1, x_2, ..., x_i, .., x_n)$, where x_i is a log template ID and n is the length of the log sequence. An embedding technique (e.g., BERT) converts each x_i to a θ-dimensional vector representing the semantics of x_i, where θ is the size of log sequence embedding. Then each log sequence forms a matrix $X \in R^{n \times \theta}$. Different log sequence embedding strategies can be applied; more information is provided in § 2.4.1.

In the classification step, the embedding matrix is processed to predict whether the given log sequence leads to a failure or not. A DL model, as an encoder Φ encodes the matrix X into a feature vector $z = \Phi(X) \in R^m$, where m is the number of features, which is a variable depending on the architecture of Φ. Different DL encoders can be applied; more information is provided in § 2.4.2. Similar to related studies [22, 38], the output feature vector z is then fed to a Feed-Forward Network (FFN) and softmax classifier to create a vector of size d ($d = 2$), capturing the prediction of the input unit label. As the FFN has a consistent setting across various configurations, it is separated as a common, trainable part of the architecture, following an architecture similar to that of the NeuralLog model [37] as well as the LogRobust one [14].

More specifically, the FFN activation function is Rectified Linear Unit (ReLu), and the output vector of the FNN r is defined as $r = max(0, zW_1 + b_1)$ where $W_1 \in R^{m \times d_f}$ and $b_1 \in R^{d_f}$ are a trainable parameter, and d_f is the dimensionality of the FNN. Further, the calculation of the softmax classifier is as follows.

$$o = rW_2 + b_2 \tag{2.2}$$

$$softmax(o_p) = \frac{exp(o_p)}{\sum_j exp(o_j)} \tag{2.3}$$

where $W_2 \in R^{d_f \times d}$ and $b_2 \in R^d$ are trainable parameters to convert r to $t \in R^d$ before applying softmax; o_p represents the p-th component in the o vector, and exp is the exponential function. After obtaining the softmax values, the position with the highest value determines the label of the input log sequence.

Overall, the configuration of an embedding strategy and a DL encoder forms a language model that takes textual data as input and transforms it into a probability distribution [27]. It is worth noting that there is no redundancy between the embedding and DL encoder components. While some embedding strategies (e.g., BERT) remain frozen and provide fixed semantic representations, others, such as Logkey2vec, are trainable yet consist of a lightweight single-layer neural network. Therefore, they do not overlap in functionality with the DL encoders, which are fully trainable and focus on capturing higher-level sequential or contextual dependencies among the embedded log templates.

To train the above architecture, several hyperparameters must be set, including the choice of optimizer, loss function, learning rate, input size (for certain deep learning models), batch size, and the number of epochs. Tuning these hyperparameters is highly recommended as it significantly increases the chances of achieving the best failure prediction accuracy. Section 2.5.2.4 will detail the training and hyperparameter tuning in our experiments.

After the model is trained, it is evaluated with a test log split from the dataset with stratified sampling. We used stratified sampling to keep the same proportion of failure log sequences as in the original dataset. Similar to training data, the embedding step transforms the test log sequences into embedding matrices. The matrices are then fed to the trained DL encoder to predict whether log sequences lead to failure or not.

2.4.1 Embedding Strategies

While the modular architecture can accommodate various log sequence embedding options, we consider only one representative instance from each of the three LSE strategies (see § 2.2.5), given our experimental constraints. More details are provided in § 2.5.2.1.

Note that the following three techniques were not compared in the same study before, according to Table 2.1.

Logkey2vec. For Logkey2vec (see § 2.2.5.1), we set the embedding size to 768, similar to BERT for better comparison. The vocabulary size is set to 200, consistent with the study of Lu [22].

BERT. The maximum number of input tokens for BERT (see § 2.2.5.2) is 512 tokens. This limit does not pose a problem in this work, as the log templates in our datasets are relatively short and the total number of tokens in each log template is always less than 512. Even if log templates were longer than 512, related studies suggest approaches to utilise BERT accordingly [65–67]. Each layer of the transformer encoder contains multi-head attention sub-layers and FFNs to compute a context-aware embedding vector ($\theta = 768$) for each token. This process is repeated for all the log templates inside the log sequence to create a matrix representation of size $n \times 768$, where n is the length of the input log sequence.

FastText+TF-IDF. Following its initial evaluation [14], the dimension of the embedding vector is set to 300 ($d = 300$).

2.4.2 Deep Learning Encoder

In this section, we illustrate the main features of the four DL encoders that can be used in the "Classification step" when instantiating our base architecture. We selected four encoders (LSTM-, BiLSTM-, CNN-, and transformer-based) because they cover the main DL types. These four encoders cover all the DL techniques used in log-based failure prediction (BiLSTM and LSTM). Additionally, they represent the most common DL techniques used in relevant log analysis tasks: LSTM has been employed in nine studies, BiLSTM and transformers in five, and CNN in three, as detailed in Table 2.1. GNNs are not included

because there is no fair way to compare them with the others due to the required pre-processing stage required to transform sequential data into graphs, which is an expensive endeavour and a subject of current research [52].

LSTM-based. This DL model is inspired by the LSTM architecture suggested by related works, including DeepLog [11], Aarohi [7], and Dash [6]. The model contains one LSTM hidden layer with 128 nodes and ReLu activation. A dropout rate of 0.1 is applied to help the model generalise better. The output of the model is a feature vector of size 128.

BiLSTM-based. The model has an architecture similar to LogRobust, which was proposed for anomaly detection. Due to its RNN-based architecture, its output is a feature vector with the same size as the input log sequence length [14].

CNN-based. The CNN architecture is a variation of the convolutional design for CNN-based anomaly detection [22]. Based on our preliminary experimental results, 20 filters, instead of one, are used in parallel for each of the three 1D convolutions (see § 2.2.4.2) to capture relationships between log templates at different distances. Padding is used to ensure that feature maps of each convolution have the same dimension as the input. Hence, the length of the output feature vector is the product of the number of filters (20), the number of convolutions (3), and the input size of the log sequence.

Transformer-based. Our architecture of the transformer model is inspired by recent work in anomaly detection [37–39]. The model consists of two main components: positional embeddings and transformer blocks. One transformer block is adopted after positional embedding, set similarly to a recent study [37]. After global average pooling, the output matrix is mapped into one feature vector of the same size as the log template embedding $\theta = 768$, previously explained in § 2.2.4.

2.5 Empirical Study Design

2.5.1 Research Questions

The goal of this study is to systematically evaluate the performance of failure predictors by instantiating our base architecture with different configurations of DL encoders and log sequence embedding strategies for various datasets with different characteristics. The ultimate goal is to rely on such analyses *to provide practical guidelines to select the right failure prediction model based on the characteristics of a given dataset.* To achieve this, we investigate the following research questions:

RQ1: What is the impact of different DL encoders on failure prediction accuracy?

RQ2: What is the impact of different log sequence embedding strategies on failure prediction accuracy?

RQ3: How do DL-based failure predictors fare compared to traditional ML-based ones in terms of failure prediction accuracy?

RQ4: What is the impact of different dataset characteristics on failure prediction accuracy?

RQ5: How does the accuracy of failure prediction on synthesised datasets compare to that of real-world datasets?

RQ1 and RQ2 investigate how failure prediction accuracy varies across DL encoders and embedding strategies reported in the literature. Most of them have been evaluated in isolation or with respect to a few alternatives, often using ad-hoc benchmarks (see § 2.3 for a detailed comparison). To address this, we comprehensively consider all variations of our base architecture, obtained by combining different DL encoders and log sequence embedding strategies that have been widely used in failure prediction and anomaly detection. Furthermore, we systematically vary the characteristics of the input datasets in terms of the number of log sequences, the length of log sequences, and the proportion of normal log sequences. The answers to these questions are expected to lead to practical guidelines for selecting the best failure prediction model, given a dataset with specific characteristics.

RQ3 compares the DL-based and traditional ML-based (also referred to as non-DL) failure predictors in terms of accuracy. This will allow us to better understand the potential advantages and drawbacks of using DL methods for failure prediction.

RQ4 further investigates the impact of input dataset characteristics on failure prediction accuracy, with a focus on the best DL encoder and log sequence embedding strategy identified in RQ1 and RQ2. The answer to this question will help us better understand under which conditions the configuration of the best DL encoder and log sequence embedding strategy works sufficiently well for practical use, possibly leading to practical guidelines on how to best prepare input datasets for increasing failure prediction accuracy.

RQ5 compares the results (in terms of failure prediction accuracy) obtained by the configuration of the best DL encoder and log sequence embedding strategy on synthetic data with those obtained on a real dataset (more details in § 2.5.2.3).

2.5.2 Methodology

As discussed in § 2.4, we can instantiate the base architecture for failure prediction with different DL encoders and log sequence embedding strategies.

To answer RQ1 and RQ2, we train different configurations of the base architecture while systematically varying the characteristics of the training datasets (e.g., size and failure types). Then, we evaluate the relative performance of the configurations in terms of failure prediction accuracy, using test datasets having the same characteristics but not

used during training. We elaborate on the different configurations, dataset characteristics, and training and testing of the failure predictor in the following sections.

To answer RQ3, we compare the results of the best configuration of the DL-based failure prediction architecture with a traditional ML-based failure predictor. We selected Random Forest (RF) as a traditional ML-based method, as it has been shown, according to a comprehensive study by Fernandez-Delgado [68], to have the best overall performance compared to other traditional ML-based methods. Moreover, in the context of log-based failure prediction, the RF-based method has shown better results compared to other traditional ML-based methods, such as XGBoost [58] and SVM [57] (see also § 2.3.1.2). Therefore, using RF provides the best insights over using DL-based failure predictors. For RF, we set the number of estimators, which is the primary hyperparameter, to 10, in line with its related study [41]. For the embedding strategy, since the input of RF is an embedding vector rather than an embedding matrix used for our modular architecture, we selected TF-IDF (template-level), the best overall embedding strategy for RF according to a close study [41].

To answer RQ4, we first identify all the top configurations, as there may be certain datasets where configurations other than the best configuration inferred from RQ1-2 perform better. We then analyse the impact of each dataset characteristic (e.g., dataset size, percentage of failure) on these configurations. To further investigate the combination of these characteristics, we construct a decision tree based on the best configuration for each dataset to predict the conditions where each top configuration fares best.

Moreover, we build regression trees [69] to automatically infer conditions describing how the failure prediction accuracy of the best configurations varies according to the dataset characteristics.

To answer RQ5, due to the limited availability of real-world datasets for failure prediction (see Section 2.3.1.2), we must choose from the available datasets for anomaly detection. Especially, datasets designed explicitly for failure detection (i.e., a sub-task of anomaly detection) are more compatible with our task, considering their transferability to failure prediction discussed in § 2.2.3.2. OpenStack is a common dataset explicitly used for failure prediction [30] and is further labeled at the log message level, which we refer to as OpenStack_v2. These characteristics enabled us to further process the data to make it suitable for failure prediction, leading to the creation of a new dataset called OpenStack_FP, which we introduce in Section 2.5.2.3. We compare the failure prediction accuracy results obtained on the synthesised datasets most similar, in terms of dataset size, failure percentage, and MLSL [1] to OpenStack_FP, with those obtained on the OpenStack_FP dataset. For practical reasons, we only focus on the accuracy results of the best DL configuration and the best traditional ML model, i.e., RF.

2.5.2.1 Log Sequence Embedding Strategies and DL Encoders

As for different log sequence embedding strategies, we considered the best-fitting instances from three categories, which have been shown to be accurate in the literature as discussed

[1] More details are provided in § 2.6.5.

in § 2.4.1. Among template ID-based strategies, we excluded the count vector since they are unable to capture sequential patterns in a log sequence (see § 2.2.5). TF-IDF methods (such as count vectors) were incompatible with our architecture because their output embedding is a vector for each log sequence, rather than a matrix. Conversely, Logkey2vec incorporates the order of log templates in the embedding procedure and yields the desired output structure. Among semantic-based strategies, since Template2vec is trained on manually added, domain-specific synonyms and antonyms, its applicability is limited, and we excluded it, as mentioned in 2.2.5.2. Among available pre-trained strategies, we included BERT, given its prevalent usage in log analysis studies and its demonstrated benefits [10, 37]. Regarding the hybrid strategy, we incorporated F+T (the aggregation of FastText with TD-IDF), a common approach in the existing literature.

As for different DL encoders in RQ1 and RQ2, we consider four encoders (LSTM, BiLSTM, CNN, and transformer) that have been previously used in related works; we describe their architecture details in § 2.4.2. We configured the encoders based on the recommendations reported in the literature (see § 2.4.2 for further details).

2.5.2.2 Datasets with Different Characteristics

As for the characteristics of datasets, we consider four factors that are expected to affect failure prediction performance: (1) dataset size (i.e., the number of logs in the dataset), (2) Log Sequence Length (LSL) (i.e., the length of a log sequence in the dataset), (3) failure percentage (i.e., the percentage of log sequences with failure patterns in the dataset), and (4) failure pattern type (i.e., types of failures).

The dataset size is important to investigate to assess the training efficiency of different DL models. To consider a wide range of dataset sizes while keeping the number of all combinations of the four factors tractable, we consider six levels that cover the range of real-world dataset sizes reported in a recent study [3]: 200, 500, 1 000, 5 000, 10 000, and 50 000.

The LSL could impact failure prediction, as a failure pattern that spans a longer log may be more challenging to predict accurately. Similar to observed lengths in real-world log sequences across publicly available datasets [3], we vary the maximum[2] LSL across five levels: 20, 50, 100, 500, and 1 000.

The failure percentage determines the balance of classes in a dataset, which may affect the performance of DL models [70]. The training dataset is perfectly balanced at 50%. However, the failure percentage can be much less than 50% in practice, as observed in real-world datasets [53]. Therefore, we vary the failure percentage across six levels: 5%, 10%, 20%, 30%, 40%, and 50%.

Regarding failure patterns, we aim to consider patterns with potential differences in terms of learning effectiveness. However, the failure patterns defined in previous studies are too simplistic; for example, Das [6] considers a specific, consecutive sequence of problematic log templates, referred to as a "failure chain". But in practice, not all problematic

[2]We set the maximum LSL for to simplify control.

log templates appear consecutively in a log. To address this, we use regular expressions to define failure patterns, allowing non-consecutive occurrences of problematic log templates. For example, a failure pattern "$x(y|z)$" indicates a pattern composed of two consecutive templates that starts with template x and ends with either template y or template z. In addition, we consider two types of failure patterns (in the form of regular expressions), *Type-F* and *Type-I*, depending on the cardinality of languages accepted by the regular expressions (*finite* and *infinite*, respectively). This is because, if the cardinality of the language is finite, DL models might memorise (almost) all the finite instances (i.e., sequences of log templates) instead of learning the failure pattern. For example, the language defined by the regular expression "$x(y|z)$" is finite since there are only two template sequences (i.e., xy and xz) matching the expression "$x(y|z)$. In this case, the two template sequences might appear in the training set, making it straightforward for DL models to simply memorise them. On the contrary, the language defined by the regular expression "$x^*(y|z)$" is infinite due to infinite template sequences that can match the sub-expression 'x^*'; therefore simply memorising some of the infinitely many sequences matching "$x^*(y|z)$" would not be enough to achieve high failure prediction accuracy.

To sum up, we consider 360 combinations (six dataset sizes, five maximum LSLs, six failure percentages, and two failure pattern types) in our evaluation. However, we could not use publicly available datasets for our experiments due to the following reasons. First, although He et al. [4] reported several datasets in their survey paper, they are mostly labelled based on the occurrence of error messages (e.g., log messages with the level of ERROR) instead of considering failure patterns (e.g., sequences of certain messages). Furthermore, there are no publicly available datasets covering all the combinations of the four factors defined above, making it impossible to thoroughly investigate their impact on failure prediction. To address this issue, we present a novel approach for synthetic log data generation in § 2.5.3.

2.5.2.3 Real-world Dataset Processing

The real-world log dataset used to address RQ5 is based on the OpenStack dataset, which is collected from a large-scale study on failures in OpenStack, as documented by Cotroneo [71]. It is known to be the most comprehensive publicly available dataset of logs including failure data generated from a cloud-based system [30], involving a wide variety of failures reported in the OpenStack bug repository[3]. Failures stem from different fault injection mechanisms (e.g., modifying the source code of OpenStack) and running a workload (task) with the injected fault. In the original OpenStack dataset, the granularity of the labels is at the level of the workload; labels are determined by checking assertions at the end of the workload runs. Bogatinovski [30] further labelled the logs at the log message level using two human annotators labelling more than 200 000 log messages to find those indicating the logged failure. We refer to this version of the dataset as OpenStack_v2. To make OpenStack_v2 ready for failure prediction, we further processed it according to the discussion on dataset transferability, as mentioned in § 2.2.3.2.

[3] `https://bugs.launchpad.net/openstack/`

Table 2.2: Overview of OpenStack_FP dataset

#Logs	#Failures Log Sequences	Failure Percentage	#Unique Log Templates	Log Sequence Length avg	min	max
876	188	21.46%	468	228	4	462

Specifically, we partition the logs according to their log identifier, which is the task ID in this context. As discussed in Section 2.2.3.1, partitioning logs using log identifiers yields higher accuracy than using timestamp-based ones. If a task ID is marked as a failure, we retain only the log messages, ordered by timestamp, up to before the occurrence of the first failure message. In this way, we eliminate the direct signs of a failure in a log, resulting in a log sequence that appears normal, although it triggers a failure. Additionally, due to the limitation on the maximum log sequence length, if a log sequence exceeds the limit, we only retain the last 1000 log messages. We set this threshold since it is the maximum input sequence length in our modular architecture; moreover, we speculate that the messages at the end of the sequence are more related to the subsequent failure. We name the processed dataset OpenStack_FP, as it is suitable for failure prediction. Table 2.2 provides a summary of the OpenStack_FP statistics, where "# logs" indicates the number of logs that form a log sequence, and "avg," "min," and "max" represent the average, minimum, and maximum lengths of the log sequences, respectively.

2.5.2.4 Failure Predictor Training and Testing

We split each artificially generated dataset, as well as OpenStack_FP, into two disjoint sets, a training set and a test set, with a ratio of 80:20. Further, 20% of the training set is separated as a validation set, which is used for early stopping [72] during training to avoid over-fitting.

For training failure predictors, to control the effect of highly imbalanced datasets, random oversampling [73] is performed on the minority class (i.e., failure logs) to achieve a 50:50 ratio of normal to failure logs in the training dataset, showing to be effective in our preliminary experiments. For all the training datasets, we use the Adam optimizer [74] with a learning rate of 0.001 and the sparse categorical cross-entropy loss function [75], considering the Boolean output (i.e., failure or not) of the models. However, we use different batch sizes and numbers of epochs for datasets with different characteristics since they affect the convergence speed of the training error (particularly the dataset size, the maximum LSL, and the failure percentage). It would, however, be impractical to fine-tune the batch size and the number of epochs for 360 individual combinations. Therefore, based on our preliminary evaluation results, we use larger batch sizes with fewer epochs for larger datasets to maintain reasonable training time without significantly compromising training effectiveness. Specifically, we set the two hyperparameters as follows:

- *Batch size*: By default, we set it to 10, 15, 20, 30, 150, and 300 for dataset sizes of 200, 500, 1 000, 5 000, 10 000, and 50 000, respectively. If the failure percentage is less than or equal to 30 (meaning more oversampling will happen to balance between normal and

Table 2.3: Overview of Hyperparameter Setting

Hyperparameter	Condition	Dataset Size					
		200	**500**	**1 000**	**5 000**	**10 000**	**50 000**
Batch Size	Default	10	15	20	30	150	300
	PF ≤ 30	10	15	30	60	300	600
	MLSL ≥ 500*	5	5	5	5	5	5
Number of Epochs	Default	20	20	20	20	20	20
	MLSL ≥ 500	20	20	10	10	5	5

* This condition has higher priority than the other.

failure logs, increasing the training data size), then we increase the batch size to 10, 15, 30, 60, 300, and 600, respectively, to reduce training time. Furthermore, regardless of the failure percentage, we set the batch size to 5 if the maximum LSL is greater than or equal to 500 to prevent memory issues during training.

- *Number of epochs*: By default, we set it to 20. If the maximum LSL is greater than or equal to 500, we reduce the number of epochs to 10, 10, 5, and 5 for dataset sizes of 1 000, 5 000, 10 000, and 50 000, respectively, to reduce training time.

Table 2.3 summarises the above conditions, where FP is the failure percentage and MLSL refers to the maximum LSL. For OpenStack_FP, we determined the hyperparameter settings by matching its characteristics to the closest ones in the table (i.e., dataset size of 1 000, failure percentage of 20%, and MLSL of 500).

Once failure predictors are trained, we measure their accuracy on the corresponding test set in terms of precision, recall, and F1 score. We also refer to robustness as a degree of consistency in accuracy in the presence of varying data set characteristics.

We conducted all experiments using cloud computing environments provided by the Digital Research Alliance of Canada [1], on the Cedar cluster, which features a total of 94 528 CPU cores for computation and 1 352 GPU devices.

2.5.3 Synthetic Data Generation

In defining a set of factors, the methodology described in § 2.5.2 makes it clear that there is a need for a mechanism that can generate datasets in a controlled, unbiased manner. Prior work on data synthesis has often utilised finite-state automata (§ 2.3.2), but they cannot accommodate the set of factors that we aim to control during synthetic data generation. For example, let us consider the factor of failure percentage (§ 2.5.2.2). Such a factor requires that one be able to control whether the log sequence being generated does indeed correspond to a failure; this would ultimately allow one to control the percentage of failure log sequences in a generated dataset.

While, for smaller datasets, one could imagine manually choosing log sequences that represent both failures and normal behaviour, for larger datasets this is not feasible. When

considering the other factors defined in § 2.5.2, such as *LSL*, the case for a mechanism for automated, controlled generation of datasets becomes yet stronger.

2.5.3.1 Key Requirements

We now describe a set of requirements that must be met by whatever approach we opt to take for generating datasets. In particular, our approach should:

R1 - Allow datasets' characteristics to be controlled. This requirement has already been described, but we summarise it here for completeness. We must be able to generate datasets for each combination of levels (of the factors defined in § 2.5.2). Hence, our approach must allow us to choose a combination of levels, and generate a dataset accordingly.

R2 - Be able to generate realistic datasets. A goal of this work is to present results that apply to real-world systems. Hence, we must require that the datasets with which we perform any evaluations reflect real-world system behaviours.

R3 - Be able to generate datasets corresponding to a diverse set of systems. While we require that the datasets that we use be realistic, we must also ensure that the data generator can generate log sequences for any system, rather than being limited to a single system.

R4 - Avoid bias in the log sequences that make up the generated datasets. For a given system, we wish to generate datasets containing log sequences that explore as much of the system's behaviour as possible (rather than being biased to a particular part of the system).

Temporal scope (timestamps). We note that the synthetic datasets generated in this study do not contain timestamps. This design choice is intentional, as our focus is on evaluating the effectiveness of failure prediction under controlled data characteristics rather than its timeliness. Incorporating timestamps would require additional assumptions about event inter-arrival times and system workload dynamics, which fall outside the scope of this work. A detailed discussion of this limitation, including its implications for practical deployment and future research on timeliness-oriented evaluation, is provided in the Threats to Validity section (§ 2.7.2) and revisited in the Future Work section (§ 4.3).

2.5.3.2 Automata for System Behaviour

Our approach is based on finite-state automata. In particular, we use automata as approximate models of the behaviour of real-world systems. We refer to such automata as

behaviour models, since they represent the computation performed by (i.e., behaviour of) some real-world system. We chose automata, or behaviour models, because some of our requirements are met immediately:

R2. Existing tools [28,29] allow one to infer behaviour models of real-world systems from collections of these systems' logs (in a process called *model inference*). Such models attach log messages to transitions, which is precisely what we need. Importantly, the collections of logs used are unlabelled, meaning that the models that we get from these tools have no existing notion of normal behaviour or failures.

R3. A result of meeting R2 is that one can easily infer behaviour models for multiple systems, provided the logs of those systems are accessible.

R4. If we are to use automata to represent systems, then we can define bias of collections of log sequences in terms of *how much* of a behaviour model is represented in those log sequences.

The remaining sections will give the complete details of our automata-based data generation approach. In presenting these details, we will show how R1 and R4 are met.

2.5.3.3 Behaviour Models

We take a behaviour model $\mathcal{M}$ to be a deterministic finite-state automaton $\langle Q, A, q_0, \Sigma, \delta \rangle$, with symbols as defined in § 2.2.1.

A behaviour model has the particular characteristic that its alphabet Σ consists of *log template IDs* (see § 2.2.2). A direct consequence of this is that one can extract log sequences from behaviour models. In particular, if one considers a sequence of states (i.e., a path) $q_0, q_i, q_{i+1}, \ldots, q_n$ through the model, one can extract a sequence of log template IDs using the transition function δ. For example, if the first two states of the sequence are q_0 and q_i, then one needs only find $s \in \Sigma$ such that $\delta(q_0, s) = q_i$, i.e., it is possible to transition to q_i from q_0 by observing s. Finally, by replacing each log template ID in the resulting sequence with its corresponding log template, one obtains a *log sequence* (see § 2.2.2). These sequences can be divided into two categories: *failure log sequences* and *normal* log sequences.

We describe failures using regular expressions. This is natural since behaviour models are finite state automata, and sets of paths through such automata can be described by regular expressions. Hence, we refer to such a regular expression as a *failure pattern*, and denote it by *fp*. By extension, for a given behaviour model $\mathcal{M}$, we then denote by $\mathsf{failurePatterns}(\mathcal{M})$ the set $\{fp_1, fp_2, \ldots, fp_n\}$ of failure patterns paired with the model $\mathcal{M}$. Based on this, we characterise *failure log sequences* as such:

Failure log sequence. For a system whose behaviour is represented by a behaviour model $\mathcal{M}$, we say that a log sequence represents a failure of the system whenever its sequence of log template IDs matches some failure pattern $fp \in \mathsf{failurePatterns}(\mathcal{M})$.

Since this definition of failure log sequences essentially captures a subset of the possible paths through $\mathcal{M}$, we define normal log sequences as those log sequences that are not failures:

Normal log sequence. For a system whose behaviour is represented by a behaviour model $\mathcal{M}$, we say that a log sequence l is normal, i.e., it represents normal behaviour, whenever $l \in \mathcal{L}(\mathcal{M})$ and $l \notin \bigcup_{fp \in \mathsf{failurePatterns}(\mathcal{M})} \mathcal{L}(fp)$ (we take $\mathcal{L}(\mathcal{M})$ and $\mathcal{L}(fp)$ to be as defined in § 2.2.1). Hence, defining a normal log sequence requires that we refer to both the language of the model $\mathcal{M}$, and the languages of all failure patterns associated with the model $\mathcal{M}$.

2.5.3.4 Generating Log Sequences for Failures

Let us suppose that we have inferred a model $\mathcal{M}$ from the execution logs of some real-world system, and that we have defined the set $\mathsf{failurePatterns}(\mathcal{M})$. Then we generate a failure log sequence that matches some $fp \in \mathsf{failurePatterns}(\mathcal{M})$ by:

1. Computing a subset of $\mathcal{L}(fp)$. We do this by repeatedly generating single members of $\mathcal{L}(fp)$. Ultimately, this leads to the construction of a subset of $\mathcal{L}(fp)$. In practice, the Python package *exrex* [76] can be used to generate random words from the language $\mathcal{L}(fp)$, so we invoke this library repeatedly.

   If the language of the regular expression is infinite, we can run *exrex* multiple times, each time generating a random string from the language. The number of runs is set based on our preliminary results regarding the range of dataset sizes (2500 times for each failure pattern). Doing this, we generate a subset of $\mathcal{L}(fp)$.

2. Choosing at random a log sequence l from the random subset $\mathcal{L}(fp)$ computed in the previous step, with $|l| \leq mlsl$ where $mlsl$ refers to the value of maximum log sequence manipulated by LSL factor. (see § 2.2.2) (*maximum LSL*, see § 2.5.2). The Python package *random* [77] was employed for this.

   We highlight that failure patterns are designed so that there is always at least one failure pattern that can generate log sequences whose length falls within this bound.

   More details on this are provided in § 2.5.4.

For requirement R4, since our approach relies on random selection of log sequences from languages generated by the *exrex* tool, we highlight that the bias in our approach is subject to the implementation of both *exrex*, and the Python package, *random*. *Exrex* is a popular package for RegEx that has more than 100k monthly downloads. Its method for generating a random matching sequence is implemented by randomly selecting choices

from the RegEx's parse tree nodes. The *Random* package utilises the Mersenne Twister algorithm [78] to generate a uniform pseudo-random number for random selection tasks.

2.5.3.5 Generating Log Sequences for Normal Behaviour

While our approach defines how failures should look using a set of failure patterns $\mathsf{failurePatterns}(\mathcal{M})$ defined over a model $\mathcal{M}$, we have no such definition of how normal behaviour should look. Instead, this is left implicit in our behaviour model. However, based on the definition of normal log sequences given in § 2.5.3.3, such log sequences can be randomly generated by performing random walks on behaviour models.

This fact forms the basis of our approach to generating log sequences for normal behaviour. However, we must also address key issues: 1) the log sequences generated by our random walk must be of bounded length, and 2) the log sequences must also lack bias.

There are two reasons for enforcing a bound on the length of log sequences:

- Deep learning models (such as CNN) often accept inputs of limited size, so we have to ensure that the data we generate is compatible with the models we use.
- One of the factors introduced in § 2.5.2 is LSL, so we need to be able to control the length of log sequences that we generate.

For bias, we have two sources: 1) bias to specific regions of the behavioural model, 2) bias to the limited variation of LSL. We must minimise bias in both cases.

Algorithm 1 gives our procedure for randomly generating a log sequence representing normal behaviour of a system. Algorithm 1 itself makes use of Algorithm 2.

Algorithm 1: *generateNormalSequence*

Input: $\mathcal{M} : behaviour\ model, mlsl : int$
Output: $sequence : \langle s_1, s_2, \ldots, s_n \rangle \in \mathcal{L}(\mathcal{M})$
1 $sequence : list \leftarrow filteredRandomWalk(\mathcal{M}, mlsl)$
2 **while** $sequence \in \bigcup_{fp_i \in \mathsf{failurePatterns}(\mathcal{M})} \mathcal{L}(fp_i)$ **do**
3 $\quad sequence \leftarrow filteredRandomWalk(\mathcal{M}, mlsl)$
4 **return** *sequence*

Once the set *options* has been computed, one transition $\langle q, s, q' \rangle$ will be chosen randomly from the set (line 13). This random choice eliminates bias because, each time we choose the next state to transition to, we do not favour any particular state (there is no weighting involved). This, extended over an entire path, means that we do not favour any particular region of a behaviour model. Now, from the randomly chosen transition $\langle q, s, q' \rangle$, the log template ID, s, is added to *sequence* (via sequence concatenation); *currentState* is updated to the next state, q'; and *maximumWalk* is decreased by one. Based on the condition of the while loop (line 5), when $currentState \in A$ (i.e., the algorithm has reached an accepting state), the generated sequence *sequence* is returned.

Algorithm 2: *filteredRandomWalk*

Input: $\mathcal{M} = \langle Q, A, q_0, \Sigma, \delta \rangle$: *behaviour model*, *mlsl* : *int*
Output: *sequence* : $\langle s_1, s_2, \ldots, s_n \rangle \in \mathcal{L}(\mathcal{M})$

```
1  sValue : int ← calculateSValues(M)
2  sequence : list ← ⟨⟩
3  maximumWalk : int ← mlsl
4  currentState : state ← q_0
5  while currentState ∉ A do
6      options : set ← ∅
7      transitions : set ← {⟨currentState, s, q⟩ : δ(currentState, s) = q}
8      for ⟨q, s, q'⟩ ∈ transitions do
9          if sValue(q') < maximumWalk then
10             options ← options ∪ {⟨q, s, q'⟩}
11     ⟨q, s, q'⟩ ← random choice from options
12     sequence ← sequence + ⟨s⟩
13     currentState ← q'
14     maximumWalk ← maximumWalk − 1
15 return sequence
```

While Algorithm 2 generates an unlabelled log sequence, Algorithm 1 generates a normal log sequence. To achieve this, the process begins by generating a log sequence through the invocation of the *filteredRandomWalk* procedure (Algorithm 2). Since the sequence generated by Algorithm 2 is unlabelled, we must ensure that we do not generate a failure log sequence. We do this by checking whether the generated log sequence, *sequence*, belongs to the language of any failure pattern in $\mathsf{failurePatterns}(\mathcal{M})$. If this is indeed the case, another sequence must be generated. This process is repeated (line 2) until the log sequence generated by the call of *filteredRandomWalk* does not match any failure pattern in $\mathsf{failurePatterns}(\mathcal{M})$. Once a failure log sequence has been generated, it is returned.

We acknowledge that this process could be inefficient (since we are repeatedly generating log sequences until we get one with the characteristics that we need). However, we highlight that failure patterns describe only a small part of a behaviour model (this is essentially the assumption that failure is a relatively uncommon event in a real system). Hence, normal log sequences generated by random walks can be generated without too many repetitions.

2.5.3.6 Correctness and Lack of Bias

We now provide a sketch proof of the correctness of Algorithm 2, along with an argument that the algorithm eliminates bias.

To prove correctness, we show that, for a behaviour model $\mathcal{M}$, the algorithm always generates a sequence of log template IDs that correspond to the transitions along a path through $\mathcal{M}$.

The algorithm begins at q_0, by setting *currentState* to q_0 (line 4). From q_0, and each successive state in the path, the possible next states must be adjacent to *currentState* (line 7). Hence, the final value of *sequence* after the while-loop at line 5 must be a sequence of log template IDs that correspond to the transitions along a path through $\mathcal{M}$.

Further, we must show that the sequence of log template IDs generated does not only correspond to a path through the behaviour model, but is of length at most *mlsl* (one of the inputs of Algorithms 2 and 1). This is ensured by three factors:

- The initialisation of the variable *maximumWalk* on line 3.
- The subsequent decrease by one of that variable each time a new log template ID is added to *sequence*.
- Filtering of the possible next states in the random walk on line 9. In particular, on line 9 we ensure that, no matter which state we transition to, there will be a path that 1) leads to an accepting state; and 2) has length less than *maximumWalk*.

Finally, bias is minimised by two factors:

- On line 13, we choose a random next state. Of course, here we rely on the implementation of random choice that we use.
- On line 9, while we respect the maximum length of the sequence of log template IDs, we do not enforce that we reach this maximum. Hence, we can generate paths of various lengths.

Example. To demonstrate Algorithm 2, we now perform a random walk over the behaviour model shown in Figure 2.5. We start with the behaviour model's starting state, q_0, with *mlsl* set to 5. Since $q_0 \notin A$, we can execute the body of the while-loop at line 5. Hence, we can determine the set *transitions* of transitions leading out of q_0:

$$\{\langle q_0, a, q_2\rangle, \langle q_0, b, q_2\rangle, \langle q_0, c, q_1\rangle, \langle q_0, d, q_1\rangle\}.$$

Our next step is to filter these transitions to ensure that the state that we move to allows us to reach an accepting state within *maximumWalk* states. To do this, we filter the set *transitions* with respect to the values in Table 2.4. After this filtering step, the resulting set, *options*, is

$$\{\langle q_0, a, q_2\rangle, \langle q_0, b, q_2\rangle, \langle q_0, c, q_1\rangle, \langle q_0, d, q_1\rangle\}.$$

All states in *transitions* are safe to transition to. To take one transition as an example, $\langle q_0, a, q_2\rangle$ has $sValue(q_2) = 1 < 5$, so is kept.

Once we have computed *options*, we choose a transition at random. In this case, we arrive at $\langle q_0, c, q_1\rangle$, meaning that we set *currentState* to q_1. Before we progress to the next iteration of the main loop of the algorithm, we also decrease *maximumWalk*. This means that, during the next iteration of the while loop, we will be able to choose transitions leading to states from which an accepting state must be reachable within less than 4 states.

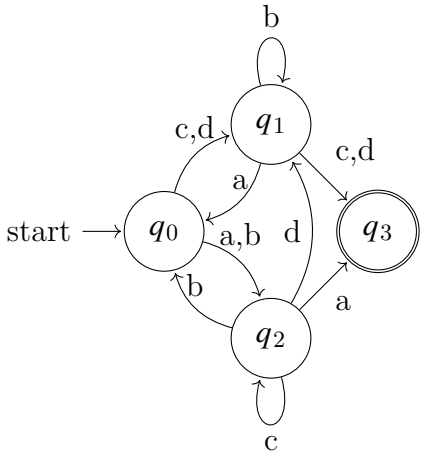


Figure 2.5: An example of a behaviour model.

Table 2.4: s values for each state

state	s value
q_0	2
q_1	1
q_2	1
q_3	0

Indeed, from q_1, there are four transitions, for which we compute the set

$$\{\langle q_1, a, q_0\rangle, \langle q_1, b, q_1\rangle, \langle q_1, c, q_3\rangle, \langle q_1, d, q_3\rangle\}.$$

From this set, each possible next state has *sValue* greater than *maximumWalk* (equal to 4), so all of them would be possible options for the next step. Suppose that we choose $\langle q_1, a, q_0\rangle$ at random. Hence, q_0 is the next state and a is added to the *sequence*. For the remaining steps, a possible run of the procedure could yield the sequence of transitions $\langle q_0, b, q_2\rangle$, $\langle q_2, d, q_1\rangle$, $\langle q_1, d, q_3\rangle$, in which case the final sequence of log template IDs would be c, a, b, d, d.

2.5.3.7 Compliance to Requirements

We now describe how the approach that we have described meets the remaining requirements set out in § 2.5.3.1.

R1 is met because we have two procedures for generating failure log sequences (§ 2.5.3.4) and normal log sequences (§ 2.5.3.5). By having these procedures, we can precisely control the number of each log sequence type in our dataset.

R4 is met because of the randomisation used in our data generation algorithm, described in Sections 2.5.3.4 and 2.5.3.5.

2.5.4 Experimental Setting for Synthesised Data Generation

To generate diverse log datasets with the characteristics described in § 2.5.2.2, using the syntactic data generation approach described in § 2.5.3, we need two main artifacts: *behaviour models* and *failure patterns*.

2.5.4.1 Behaviour Models

Regarding behaviour models, as discussed in § 2.5.3.2, we can infer accurate models of real-world systems from their execution logs using state-of-the-art model inference tools, i.e., MINT [29] and PRINS [28]. Among the potential models we could generate using the replication package of these tools, we choose models that satisfy the following criteria based on the model size and inference time reported by Shin [28]:

(1) The model should be able to generate (accept) a log with a maximum length of 20 (i.e., the shortest maximum LSL defined in § 2.5.2.2;

(2) Since there is no straightforward way to automatically generate failure patterns for individual behaviour models considering the two failure pattern types, we had to manually generate failure patterns (detailed in § 2.5.4.2). Therefore, the size of the model should be amenable to manually generating failure patterns by taking into account the model structure (i.e., the number of all states and transitions is less than 1 000);

(3) The model inference time should be less than 1 hour; and

(4) If we can use both PRINS and MINT to infer a model that satisfies the above criteria for the same logs, then we use PRINS, which is much faster than MINT in general, to infer the model.

As a result, we use the following three models as our behaviour models: $\mathcal{M}_1$ (generated from NGLClient logs using PRINS), $\mathcal{M}_2$ (generated from HDFS logs using MINT), and $\mathcal{M}_3$ (generated from Linux logs using MINT). Specifically, the NGLClient dataset was provided by the PRINS authors and collected from a Personal Computer (PC) running desktop business applications on a daily basis [28], while the HDFS and Linux datasets were obtained from the widely used LogHub repository [79]. The three datasets represent different types of systems: NGLClient from end-user desktop applications, HDFS from a large-scale distributed file system, and Linux from an operating system kernel. Table 2.5 reports on the size of the three behaviour models in terms of the number of templates (#Templates), average length of templates (Avg Template Length) using a white-space separator, the number of states (#States), and transitions (#Transitions). It additionally shows the number of states in the largest strongly connected component (#States-NSCC) [80], which indicates the complexity of a behaviour model (the higher, the more complex). We remark that the log data were provided by the PRINS authors, who had already preprocessed them using state-of-the-art parsers (Drain [81] and MoLFI [82]) and further refined them manually to ensure the quality of the templates. We also note that the number of templates refers to the number of unique templates preserved in the fine-state automaton, which can differ from the total number of unique log templates in the original log data.

Table 2.5: Overview of Behavioural models

Model	#Templates	Avg Template Length	#States	#Transitions	#States-NSCC
$\mathcal{M}_1$	70	54	154	195	5
$\mathcal{M}_2$	16	51	91	189	72
$\mathcal{M}_3$	115	39	350	486	331

2.5.4.2 Failure Patterns

Regarding failure patterns, recall a failure pattern *fp* of a behaviour model $\mathcal{M}$ is a regular expression where $\mathcal{L}(fp) \subset \mathcal{L}(\mathcal{M})$, as described in § 2.5.3.3. Also, note that we need two types of failure patterns (*Type-F* and *Type-I*), and the failure log sequences generated from the failure patterns must satisfy the dataset characteristics (especially the maximum LSL) defined in § 2.5.2.2. To manually create such failure patterns (regular expressions) in an unbiased way, we used the following steps for each behaviour model and failure pattern type:

Step 1: We randomly choose the alphabet size of a regular expression and the number of operators (i.e., alternations and Kleene stars; the latter is not used for *Type-F*).

Step 2: Using the chosen random values, for a given behaviour model $\mathcal{M}$, we manually create a failure pattern (regular expression) *fp* to satisfy $\mathcal{L}(fp) \subset \mathcal{L}(\mathcal{M})$ and the maximum LSL within the time limit of 1 hour; if we fail (e.g., if the shortest log in $\mathcal{L}(fp)$ is longer than the maximum LSL of 20), we go back and restart Step 1.

Step 3: We repeat Steps 1 and 2 ten times to generate ten failure patterns and then randomly select three out of them.

As a result, we use 18 failure patterns (i.e., 3 failure patterns × 3 behaviour models × 2 failure pattern types) for synthetic data generation. Table 2.6 reports the characteristics of failure patterns in terms of their behavioural model (Model), pattern type (Type), the length of the pattern in terms of letters and operators (Length), size of the alphabet (#Alphabet), and the number of operators (#Operators). Additionally, it includes the maximum depth of Kleene Star Structure(s) for *Type-I* (Star Depth), which indicates the maximum depth of a nesting structure (e.g., 3 for $((b^*c)^*a)^*b$). Since there are three failure patterns per behavioural model and type, their values are presented in the form of a triple, respectively. For instance, under the $\mathcal{M}_2$ model and *Type-I*, the first failure pattern has a length of 31 and an alphabet size of 5, uses 10 operators, and showcases a star depth of 1. While the complexity of failure patterns is bounded by their behavioural model (see § 2.5.3.4), there is a wide variability among failure patterns across each characteristic.

2.5.4.3 A Remark on Generalisability.

At this point, we highlight that we cannot give failure patterns that are representative of real-world patterns for two key reasons:

Table 2.6: Overview of Failure Patterns

Model	Type	Length	#Alphabet	#Operators	Star Depth
$\mathcal{M}_1$	F	(14, 34, 35)	(7, 17, 30)	(5, 8, 1)	-
	I	(17, 25, 41)	(16, 15, 16)	(1, 7, 8)	(1, 2, 2)
$\mathcal{M}_2$	F	(20, 27, 32)	(11, 9, 11)	(3, 6, 7)	-
	I	(31, 8, 39)	(5, 5, 12)	(10, 1, 9)	(1, 1, 2)
$\mathcal{M}_3$	F	(134, 36, 48)	(77, 16, 14)	(16, 10, 5)	-
	I	(44, 30, 124)	(11, 16, 78)	(12, 7, 13)	(1, 2, 1)

- Failure patterns are necessarily dependent on the behaviour model, itself representing a specific system.
- To the best of our knowledge, there are no failure patterns derived from real-world systems reported in the literature.

Hence, instead of aiming to generate a set of failure patterns that are somehow representative of a target that is necessarily elusive, we aim to work with failure patterns that are *diverse*.

We ensure this by first separating failure patterns into two types: *Type-F* and *Type-I*. Distinguishing between failure patterns that accept infinite and finite languages enables us to observe how our failure prediction machinery performs when the language of log sequences to work with is infinite versus finite.

Second, in varying the alphabet size, we control how much of a behaviour model a failure pattern can capture. Hence, across ten randomly generated failure patterns, we would generate failure patterns that could explore only a small region of the behaviour model, along with others that would explore larger regions of the behaviour model.

Further, in varying how many (if any) alternations are used, we control how many *selections* can be made when traversing a behaviour model. For example, the failure pattern $a \mid b$ allows a single selection; we either take the a transition, or the b transition. However, the failure pattern $(a \mid b)(c \mid d)$ allows two selections; we first either take a or b, and then we either take c or d.

Finally, in varying how many (if any) Kleene stars are used, we control how many opportunities for *cycles* we have when traversing our behaviour model. For example, if we have a^*, then we have a single opportunity to loop on a. If we have $(a \mid b)^*(c \mid d)^*$, then we have two opportunities to loop: on either a or b, and then on either c or d.

As a result, the various elements of control that we introduced above lead to the selection of failure patterns that will generate a large variety of log sequences from a single behaviour model.

2.5.4.4 Overview of Synthesised Data.

As the correctness of the synthetic data generation was discussed in § 2.5.3, the synthesised datasets should follow the desired characteristics we specified in § 2.5.2.2. Here we present an overview of additional statistics regarding the dataset generation. Figure 2.6 summarises three statistics in terms of average and minimum length of log sequences (Subfigure 2.6a and Subfigure 2.6b) as well as the number of unique log templates in each dataset (Subfigure 2.6c). Each box is based on 360 datasets generated from its corresponding behavioural model $\mathcal{M}_1$, $\mathcal{M}_2$, or $\mathcal{M}_3$, where we denote the collection of all datasets generated from $\mathcal{M}i$ as $\mathcal{D}_{\mathcal{M}_i}$. A boxplot representation is used: the box shows the Interquartile Range (IQR), which is the distance between the first quartile (25^{th} percentile) and the third quartile (75^{th} percentile), thus capturing the middle 50% of the data. The whiskers extend to capture variability outside this range, the horizontal line indicates the median (50^{th} percentile), and triangles mark the mean values.

We note that log sequences within synthesised datasets are directly generated by our approach introduced in 2.5.3. Thus, no partitioning method is needed. However, given that each log sequence simulates a complete walk (meaning from the initial state to the accepting state of a behavioural model), it more closely resembles the log sequences partitioned based on the log identifier. Based on Subfigure 2.6a, $\mathcal{D}_{\mathcal{M}_3}$, synthesised datasets from $\mathcal{M}_3$, exhibit the largest Interquartile Ranges (IQRs), indicating a significant variation in average log sequence length. This arises from the higher complexity of $\mathcal{M}_3$ in terms of the number of states and templates (see § 2.5.4.1). Based on Subfigure 2.6b, the minimum length of log sequences remains relatively consistent across models. In Subfigure 2.6c, $\mathcal{D}_{\mathcal{M}_1}$ and $\mathcal{D}_{\mathcal{M}_2}$ closely align with the number of unique templates reported for $\mathcal{M}_1$ and $\mathcal{M}_2$ in Table 2.5. $\mathcal{D}_{\mathcal{M}_3}$, however, shows a large IQR, ranging from the number of unique log templates in $\mathcal{M}_3$, 115, to as few as 35. Given that MLSL values (refer to Algorithm 2) can be as low as 20, our algorithm is constrained to reach an accepting state within a specified number of transitions followed during a walk on the behavioural model, defined as a finite state automaton. As a result, not all transitions in the automaton are guaranteed to be covered. Consequently, the log templates associated with the uncovered transitions are not present in the generated log sequence. Overall, the statistics of the synthesised datasets are consistent with our settings.

2.6 Results

This section presents the results of RQ1 (DL encoders), RQ2 (log sequence embedding strategies), RQ3 (traditional ML), RQ4 (dataset characteristics), and RQ5 (real-world data), respectively.

2.6.1 RQ1: DL Encoders

Figure 2.7 shows boxplots of the failure prediction accuracy (F1 score) for different DL encoders (i.e., transformer-based, LSTM-based, CNN-based, and BiLSTM-based models)

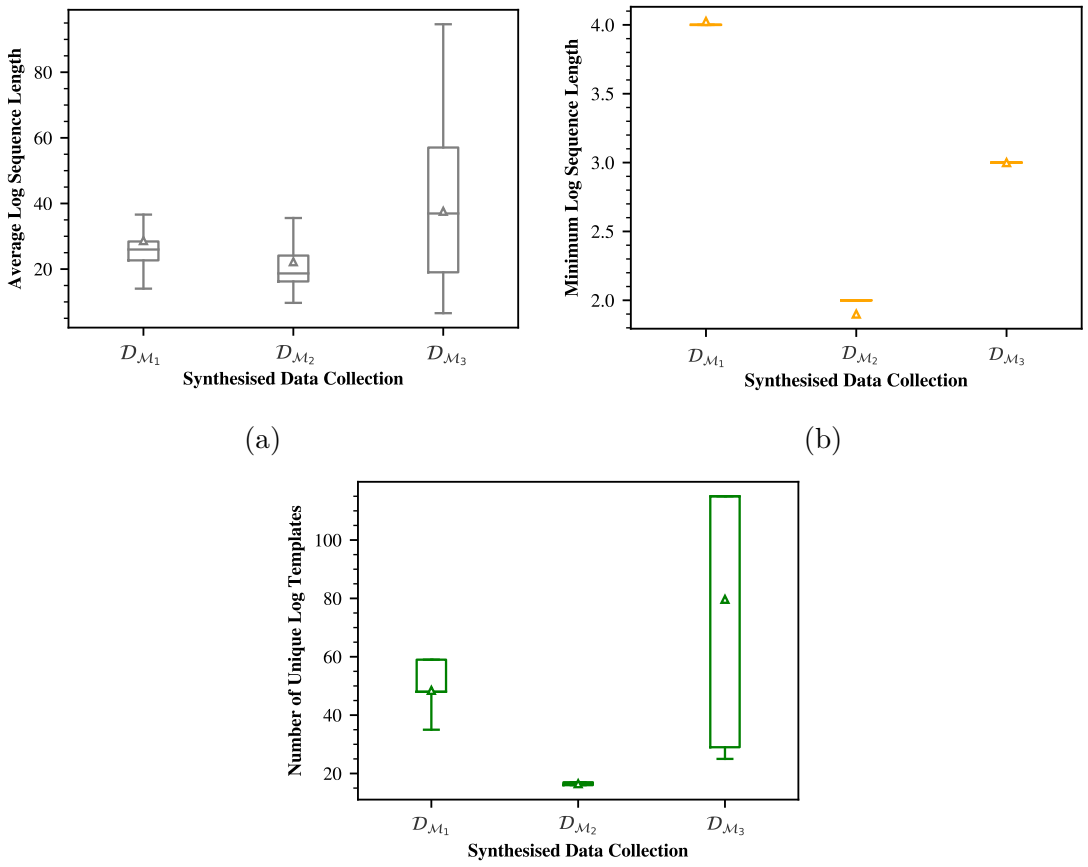


Figure 2.6: Overview of Synthesised Datasets. The triangles indicate mean values.

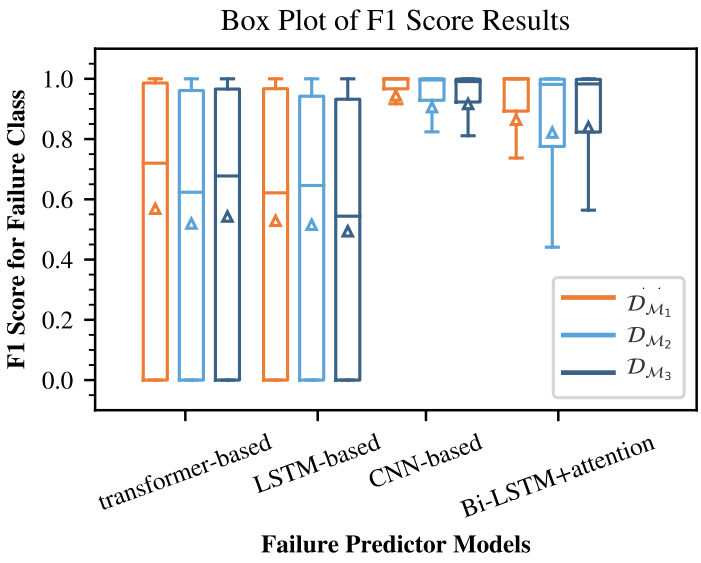


Figure 2.7: Failure prediction accuracy for different DL encoders. The triangles additionally indicate mean values.

on the datasets generated by different behaviour models (i.e., $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$). Each box is generated based on 360×3 data points since we have 360 combinations of dataset characteristics and three log sequence embedding strategies. In each box, a triangle indicates the mean value.

Overall, the CNN-based model achieves the best performance in terms of F1 score for all behaviour models. It has the highest mean values with the smallest interquartile ranges (IQRs; see § 2.5.4.4), meaning that the CNN-based model consistently works very well regardless of dataset characteristics and log sequence embedding strategies. The BiLSTM-based model also shows promising results. However, the CNN-based model's results are significantly higher for all the behavioural models (paired Wilcoxon test p-values $\ll 0.001$). In contrast, the LSTM-based and transformer-based models yield poor results, with an average low F1 score and very large interquartile ranges (IQRs). These patterns are independent from both the embedding strategy and the model. Further, the large IQR for LSTM-based and transformer-based models suggests that these models are very sensitive to the dataset characteristics.

The poor performance of the transformer-based encoder can be explained by the fact that the transformer blocks in the encoder are data-demanding (i.e., requiring much training data). When the dataset size is small (below 1 000), the data-demanding transformer blocks are not well-trained, leading to poor performance. This limitation is thoroughly discussed in the literature [83].

The LSTM-based encoder, on the other hand, has two simple layers of LSTM units. Recall that an LSTM model sequentially processes a given log sequence (i.e., a sequence of templates), template by template. Although LSTM attempts to address the long-term dependency problem of RNN by having a *forget gate* (see § 2.2.4.1), it is still a recurrent

network that has difficulties remembering a long input sequence [84]. For this reason, since our log datasets contain long log sequences (up to a length of 1 000), the LSTM-based encoder did not work well.

The BiLSTM-based encoder involves LSTM units and therefore has the weakness mentioned above. However, for BiLSTM, the input sequence flows in both directions in the network, utilizing information from both sides. Furthermore, it is enhanced by the attention mechanism that assigns more weight to parts of the input that are associated with the failure pattern [27]. Thus, the BiLSTM-based encoder can more easily learn the impact of different log templates on the classification results. However, the attention layer is more data-demanding than the convolution layers (see § 2.4.2) in the CNN-based encoder, and this explains why the BiLSTM-based encoder does not outperform the CNN-based encoder.

The high performance of the BiLSTM-based and CNN-based encoders can be attributed to the number of trainable parameters; for these two encoders, unlike the transformer-based and LSTM-based ones, the number of trainable parameters increases as the input sequences become longer. The larger number of parameters makes the encoders more robust to longer input log sequences. Furthermore, CNN additionally processes spatial information (i.e., how templates relate to each other in the data) using multiple filters with different kernel sizes [85], which makes failure prediction more accurate even when the input size (sequence length) is large. These characteristics make the CNN-based encoder the best choice in our application context.

The answer to RQ1 is that the CNN-based encoder tends to significantly outperform the other encoders across the range of data characteristics and sequence embedding strategies.

2.6.2 RQ2: Log Sequence Embedding Strategies

Figure 2.8 shows the boxplots of the failure prediction accuracy (F1 score) for the different log sequence embedding strategies considered in this study (i.e., BERT, F+T, and Logkey2vec) on the datasets generated by the three behaviour models ($\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$). Each box is generated based on 360×4 data points since we have 360 combinations of dataset characteristics and four DL encoders. Similar to Figure 2.7, the triangle in each box indicates the mean value. We now inspect the plots shown inside to answer our research questions. The plots based on precision and recall are excluded since they draw similar conclusions.

Figure 2.8 shows that the BERT embedding strategy performs better than F+T and Logkey2vec for all behaviour models in terms of mean values and smaller IQRs. This means that, on average, for all DL encoders, the semantic-aware log sequence embedding using BERT outperforms both F+T, which employs FastText but lacks the informativeness of BERT, and Logkey2vec, which relies solely on log template IDs and does not account for the semantic information of templates.

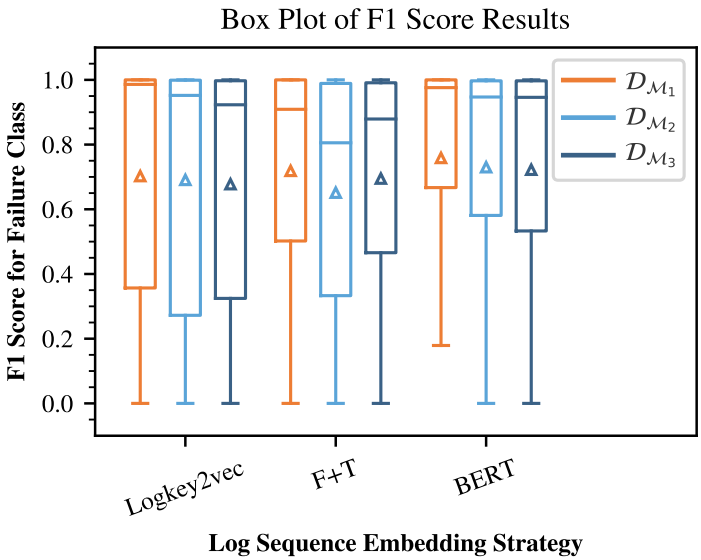


Figure 2.8: Failure prediction accuracy for different log sequence embedding strategies. The triangles additionally indicate mean values.

Table 2.7: Friedman test results (p-values). A level of significance $\alpha = 0.01$ is used. In case of a significant difference, the best strategy is denoted as l (Logkey2vec), f (F+T), and b (BERT).

DL encoder	$\mathcal{D}_{\mathcal{M}_1}$	$\mathcal{D}_{\mathcal{M}_2}$	$\mathcal{D}_{\mathcal{M}_3}$	**All**
CNN	$\ll 0.001$ (B)	$\ll 0.001$ (L)	$\ll 0.001$ (L)	$\ll 0.001$ (L)
BiLSTM	$\ll 0.001$ (B)	$\ll 0.001$ (B)	0.001 (B)	$\ll 0.001$ (B)
transformer	0.068	$\ll 0.001$ (F, B)	$\ll 0.001$ (F, B)	$\ll 0.001$ (F, B)
LSTM	$\ll 0.001$ (B)	$\ll 0.001$ (L, B)	$\ll 0.001$ (L)	$\ll 0.001$ (B)
All	$\ll 0.001$ (l)	$\ll 0.001$ (L, B)	$\ll 0.001$ (B)	$\ll 0.001$ (B)

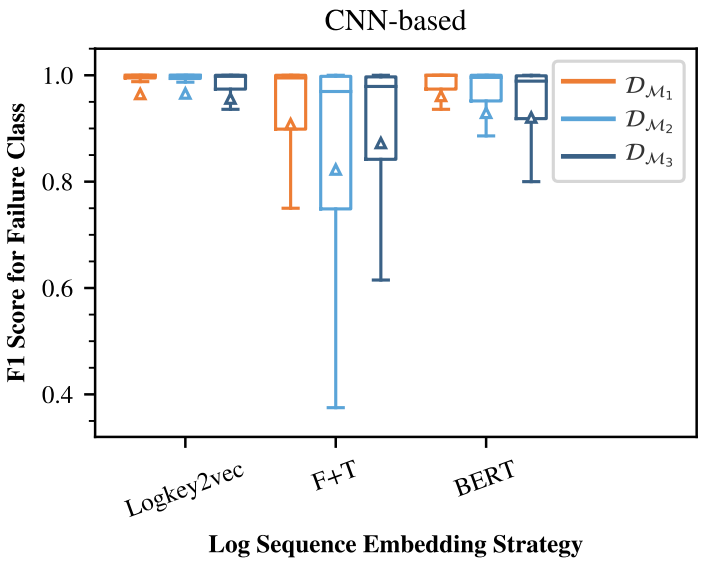


Figure 2.9: Failure prediction accuracy of CNN-based model for different log sequence embedding strategies; triangles indicate mean values.

To better understand the impact of log sequence embedding strategies on the performance of different DL encoders, we additionally performed Friedman test as a non-parametric test to compare the F1 score distributions of BERT, F+T, and Logkey2vec for each of the four DL encoders. Table 2.7 reports the statistical test results. For example, the low p-value in column $\mathcal{D}_{\mathcal{M}_2}$ and row *CNN* indicates that there are statistically significant differences between embedding strategies. In such cases, we employ a paired Wilcoxon test between each pair of embedding strategies and compare their medians to identify the top-performing strategy(ies). These are represented between brackets as L (Logkey2vec), F (F+T), and B (BERT) in the Table.

Interestingly, BERT is statistically better than or equal to F+T and Logkey2vec for all DL encoders except the CNN-based encoder (i.e., the best-performing DL encoder as investigated in § 2.6.1) and the LSTM-based encoder for $\mathcal{D}_{\mathcal{M}_3}$. On the other hand, for the CNN-based encoder, the best overall embedding strategy is Logkey2vec, as clearly observable in Figure 2.9, depicting the F1 score distributions of Logkey2vec, F+T, and BERT for the CNN encoder. In other words, combining the CNN-based encoder and the Logkey2vec embedding strategy is the best configuration of DL encoders and log sequence embedding strategies. Although, in contrast to BERT, Logkey2vec does not consider the semantic information of log templates, it accounts for the order of template IDs in each log sequence. Furthermore, Logkey2vec is trained in conjunction with the DL encoder, whereas BERT is pre-trained independently of the DL encoder. We suspect that such characteristics of Logkey2vec play a positive role when combined with the CNN-based encoder. F+T presents the largest IQR and lowest mean and median. This observation is consistent with the overall strategy comparison depicted in Fig 2.8, and similar rationales apply.

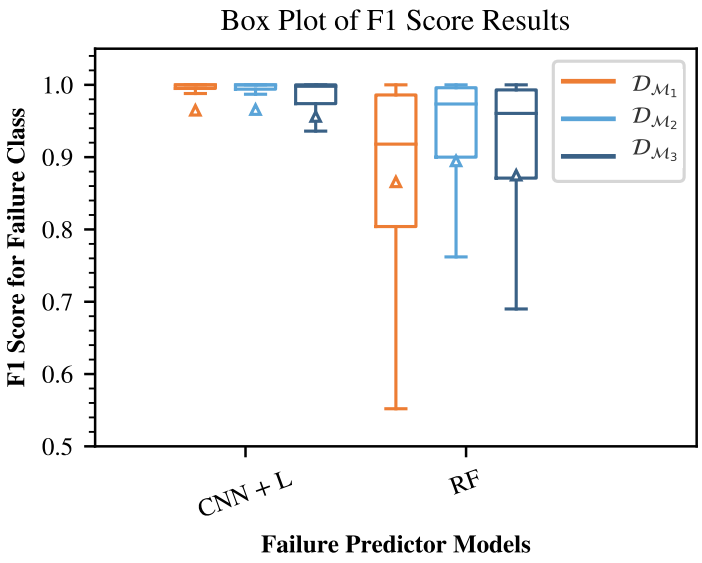


Figure 2.10: Failure prediction accuracy of the best DL-based configuration (CNN with Logkey2vec) next to a traditional ML-based configuration (RF); triangles depict mean values.

We note that BERT remains an attractive strategy for log sequence embedding when any encoder other than CNN is used. Although BERT is considerably larger than Logkey2vec in terms of parameters, using BERT does not require significantly more time and resources than Logkey2vec and F+T since BERT minimises repeated calculations by mapping each log template to its corresponding BERT embedding vector.

The answer to RQ2 is that the performance of the log sequence embedding strategies varies depending on the DL encoders used. Although BERT outperforms F+T and Logkey2vec overall across all encoders, Logkey2vec outperforms BERT when the CNN-based encoder is used.

2.6.3 RQ3: Traditional ML

Figure 2.10 shows the boxplots of the failure prediction accuracy (F1 score) for the best configuration of the DL encoder and the log sequence embedding strategy, i.e., the CNN-based encoder and Logkey2vec, next to one of the best performing [57, 58, 68], traditional ML-based failure predictor (RF), on the datasets generated by the three behaviour models ($\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$). Each box is generated based on 360 data points since we have 360 combinations of dataset characteristics. Similar to the previous boxplots, the triangle in each box indicates the mean value. We shall now examine the provided plots, aiming to address our research question.

In Figure 2.10, the CNN-based encoder with Logkey2vec clearly achieves significantly higher accuracy and robustness compared to RF, in terms of average accuracy and IQR,

respectively, regardless of the behaviour models used to generate the log datasets. RF relies on aggregating decisions from multiple trees, which can limit its ability to capture intricate, non-linear patterns of failures. In contrast, CNNs, as described in § 2.2.4.2, use convolutional layers to automatically extract hierarchical features from embedded representations, combined with pooling layers that reduce spatial dimensions, allowing CNNs to handle more complex patterns. Additionally, as explained in § 2.5.2, the input of RF is an embedding vector rather than an embedding matrix using TF-IDF, which is a template-ID-based strategy. The best DL configuration also uses a template ID-based strategy, Logkey2vec. However, unlike TF-IDF, Logkey2vec embeddings keep updating during failure predictor training; this enables Logkey2vec to learn the embeddings with respect to labels of the log sequences, see § 2.2.5.1.

The answer to RQ3 is that, using the best configuration of the DL-based failure predictor, i.e., the CNN-based encoder and Logkey2vec, results in significantly higher accuracy and robustness (low IQR) compared to Random Forest, which is considered one of the top traditional ML classifiers.

2.6.4 RQ4: Dataset Characteristics

Recall that there are 12 possible configurations for the DL-based architecture (i.e., four DL encoders and three embedding strategies), each of which may exhibit varying performances across different data set characteristics. Although CNN+L (CNN-based encoder with Logkey2vec) is the best configuration overall based on RQ1 and RQ2 results, there may be datasets where other configurations fare better. Therefore, it could potentially be informative to investigate each of the configurations in terms of their accuracy for different dataset characteristics. However, many configurations clearly provide low accuracy for most of the datasets and do not significantly outperform the other cases. So we first determined the best configurations worth investigating across the 1080 datasets. Specifically, for each configuration, we counted the number of datasets for which that configuration is among the best. We defined a threshold r set to 0.01 to include all configurations with a difference in accuracy value lower than the threshold r. This way, we could account for all high-performing configurations. It turned out that only the following three configurations kept appearing among the best configurations for almost all datasets[4]: CNN+L (CNN encoder with Logkey2vec), CNN+B (CNN-based encoder with BERT), and BiLSTM+B (BiLSTM-based encoder with BERT). Note that the top three configurations remained the same for different threshold values ($r = 0, 0.05, 0.1$). Table 2.8 provides more details about the three best configurations; column "#Best ($r = 0.01$)" provides the number of instances where the confuguration is among the best, and additional columns "Avg", "Med", "Min", and "Max" show the average, median, minimum, and maximum F1 scores for the configurations, respectively. Based on the above observations, we focus our analysis of dataset

[4]There were only 72 out of 1080 datasets where the configurations other than the top three configurations were among the best. However, not only these were very rare but their accuracy was too low to be useful.

Table 2.8: Overview of the Three Best Configurations for DL-based Failure Prediction

Rank	#Best ($r = 0.01$)	Config	Avg	Med	Min	Max
1	866	CNN+L	0.962	1.0	0.0	1.0
2	667	CNN+B	0.936	0.997	0.0	1.0
3	627	BiLSTM+B	0.879	0.995	0.0	1.0

characteristics on the three best configurations, while providing the same plots for the rest of the configurations as supplementary material in our replication package (see § 2.6.6).

Figure 2.11 shows the distributions of F1 scores according to different dataset characteristic values for CNN+L, the best configuration overall. To save space, we have excluded the plots for the second-best and third-best configurations from the paper as they were very similar to CNN+L, except for the maximum sequence length for BiLSTM+B, which will be discussed separately later. However, all the remaining plots can be found in our replication package, as previously mentioned. We discuss next how the failure prediction accuracy of CNN+L varies with each of the dataset characteristics.

In Figure 2.11a, we can see the impact of dataset size on the failure prediction accuracy; it is clear that accuracy decreases with smaller datasets, regardless of the behaviour models used to generate the log datasets. For example, when the dataset size is 200, accuracy decreases below 0.7 in the worst case, whereas it always stays very close to 1.0 when the dataset size is above or equal to 5 000. Since larger datasets imply more training data, this result is intuitive but it clarifies data requirements for failure prediction.

Figure 2.11b depicts the impact of maximum LSL values (*MLSL*) on the failure prediction accuracy. Compared to the impact of data set size, we can see that its impact is relatively small. This implies that CNN+L works fairly well for long log sequence lengths of up to 1 000. We suspect that the impact of log sequence length could be significant for much longer log sequences. However, log sequences longer than 1 000 are not common in publicly available, real-world log datasets [3] as explained in Section 2.5.2.2. Nevertheless, investigating much longer log sequences would be informative.

The relationship between failure percentage and failure prediction accuracy (F1 score) is depicted in Figure 2.11c. It is clear that, overall, the F1 score increases as the failure percentage increases. This is intuitive since a larger failure percentage means more instances of failure patterns in the training data, making it easier to learn such patterns. An interesting observation is that the average failure prediction accuracy is above 0.9 even when the failure percentage is 10%. This implies that CNN+L can cope well with unbalanced data.

Figure 2.11d shows the failure prediction accuracy for different failure pattern types. There is no consistent trend across data collections of $\mathcal{D}_{\mathcal{M}_1}$, $\mathcal{D}_{\mathcal{M}_2}$, and $\mathcal{D}_{\mathcal{M}_3}$; *Type-F* (the corresponding language is finite) is easier to detect than *Type-I* (the corresponding language is infinite) in $\mathcal{D}_{\mathcal{M}_2}$ and $\mathcal{D}_{\mathcal{M}_3}$, whereas the opposite happens in $\mathcal{D}_{\mathcal{M}_1}$. It is unclear why, in $\mathcal{M}_1$, detecting less complex failure patterns (*Type-F*) is more difficult than detecting more complicated patterns (*Type-I*). We may not have defined failure pattern types in a way

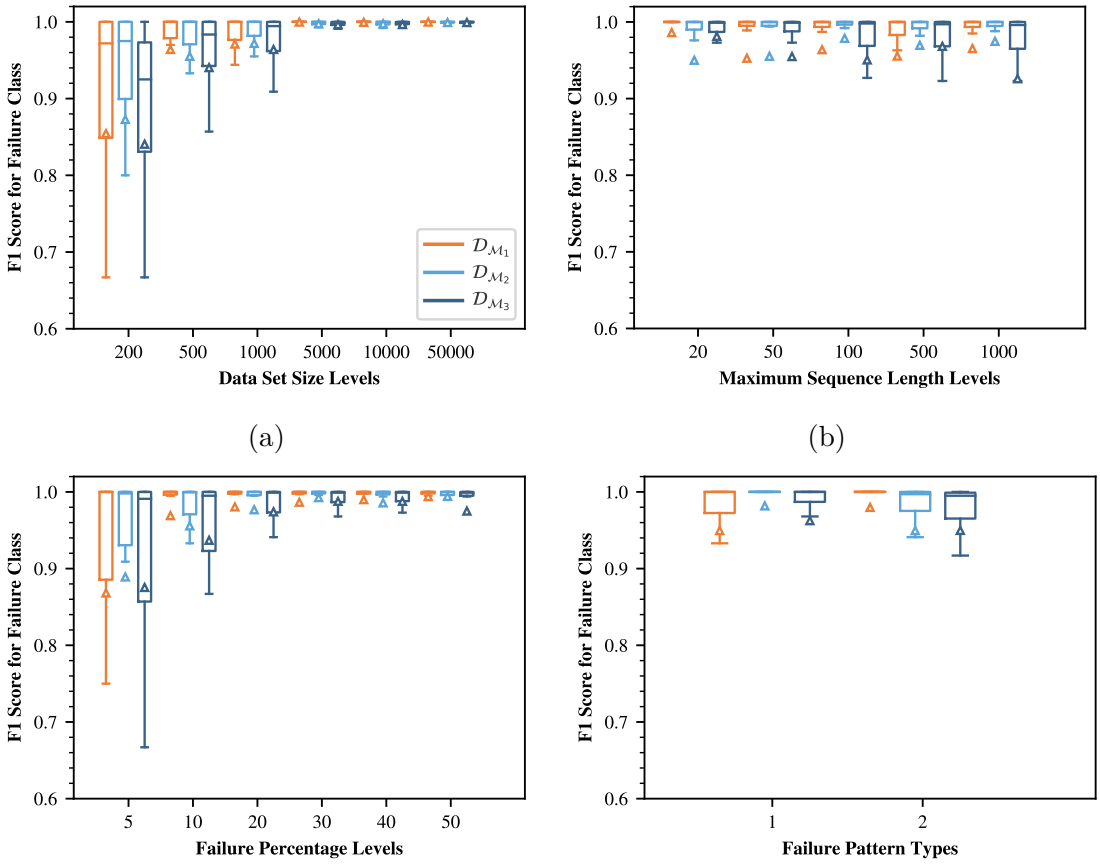


Figure 2.11: Failure prediction accuracy of the CNN-based encoder with Logkey2vec for different dataset characteristics

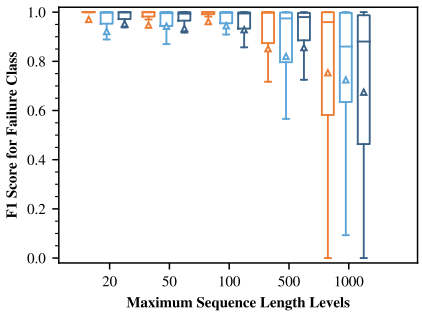


Figure 2.12: Failure prediction accuracy of the BiLSTM-based encoder with BERT as a function of maximum sequence length

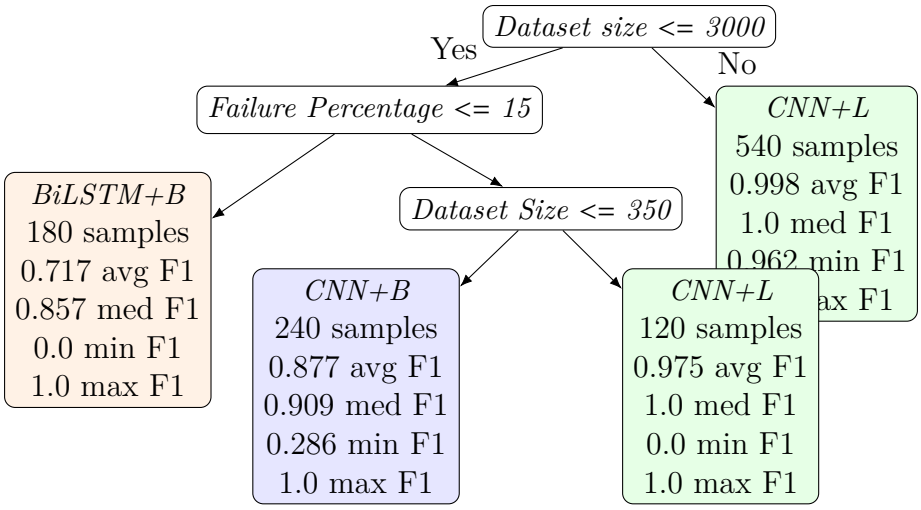


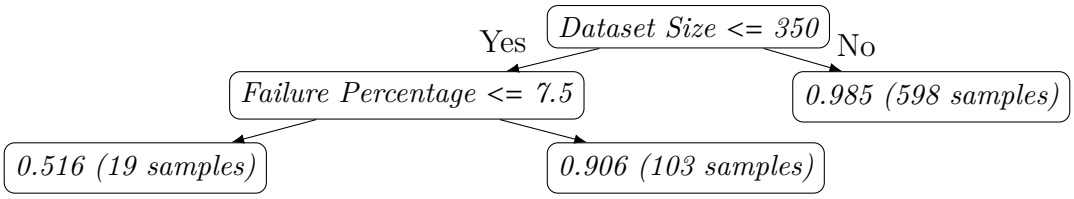
Figure 2.13: Decision Tree identifying the best configurations based on dataset characteristics

that is conducive to explaining variations in accuracy, and different hypotheses will have to be tested in future work with respect to which pattern characteristics matter.

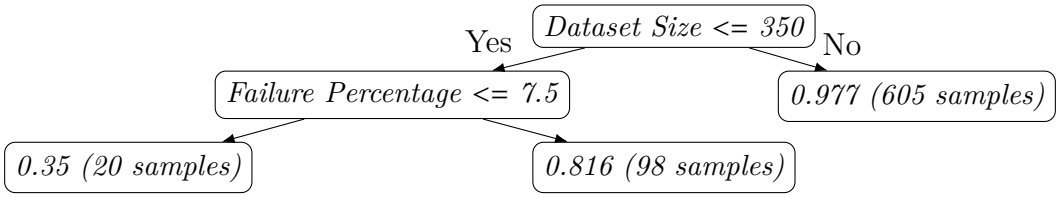
As mentioned earlier, BiLSTM+B shows a distinct result only for longer log sequences, as depicted in Figure 2.12. Unlike CNN+L shown in Figure 2.11(b), larger IQR and lower average values are clearly visible for longer log sequences in Figure 2.12. This indicates that significantly increasing the maximum length of log sequences decreases the failure prediction accuracy of BiLSTM+B.

To further investigate the data set characteristics that work well with the three best configurations, we built a classification tree predicting the best configuration for given dataset characteristics. For this purpose, we labelled each of the 1080 datasets with the configuration that achieved the highest F1 score among the three, thereby assigning the top-performing configuration as the dataset's label. We then split the 1080 datasets into subsets of 720 (66.7%) and 360 (33.3%) datasets for training and testing the classification tree, respectively. Since the training data was imbalanced due to the superior performance of CNN+L for most datasets, we applied higher weights to minority classes using Inverse Proportional Weighting [86] to address the class imbalance issue. We also performed Minimal Cost-Complexity Pruning (MCCP) [87] to avoid over-fitting. Figure 2.13 shows the resulting classification tree, where each non-leaf node captures a decision condition and each leaf node the (predicted) best configuration for the conditions corresponding to the path from the root to the leaf. Each leaf node also includes the number of samples in the leaf, as well as the average ("avg"), median ("med"), minimum ("min"), and maximum ("max") F1 score for the predicted configuration. For example, the rightmost leaf node indicates that CNN+L is the best configuration when the dataset size exceeds 3000. A total of 540 of the 1080 datasets satisfy this condition, and we can expect a failure prediction accuracy of 0.998 when using CNN+L.

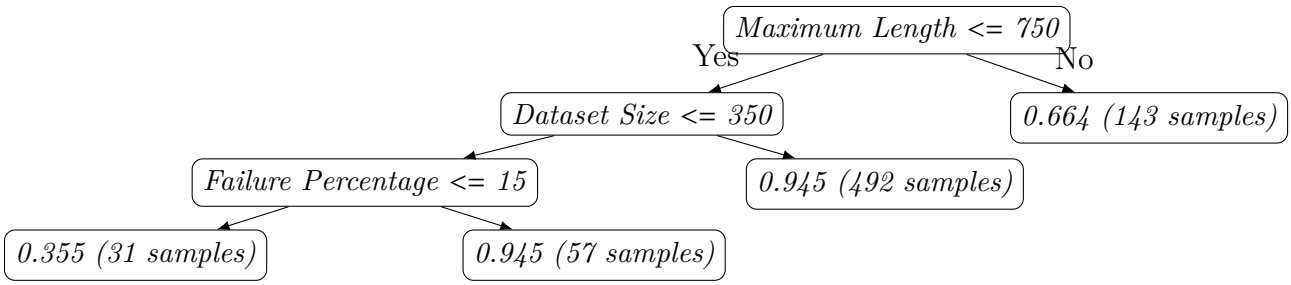
The classification tree shows that dataset size and, to a lesser extent, failure percentage play a pivotal role in determining the best configuration for the DL-based failure predic-



(a) CNN and Logkey2vec (CNN+L)



(b) CNN and BERT (CNN+B)



(c) BiLSTM and BERT (BiLSTM+B)

Figure 2.14: Regression Tree for the best configurations based on dataset characteristics in F1 scores

tor. Specifically, CNN+L is recommended for dataset sizes larger than 3000. However, for smaller dataset sizes, if the failure percentage is lower than or equal to 15%, BiLSTM+B is the recommended configuration. In other words, for dataset sizes of 3000 or less and failure percentages of 15% or less, BiLSTM+B performs better than CNN+L and CNN+B. We suspect this result is due to BiLSTM+B's higher capability in the presence of highly imbalanced datasets. In contrast, if the failure percentage exceeds 15%, CNN+B is recommended for datasets of 350 or fewer elements, while CNN+L is recommended for datasets with a size exceeding 350. In other words, for dataset sizes lower than or equal to 3000 and failure percentages higher than 15%, CNN+B performs the best. This can be attributed to the challenges posed by a small dataset for training Logkey2vec from scratch, which leads to better, semantic-enabled embeddings from BERT.

We additionally built regression trees for each of the three best configurations to further investigate how their failure prediction accuracy varies according to dataset characteristics. We applied the same approach used for pruning the classification tree above.

Figure 2.14 depicts the regression trees for CNN+L (Figure 2.14a), CNN+B (Fig-

ure 2.14b), and BiSLTM+B (Figure 2.14c). For example, in Figure 2.14a, the left-most leaf node indicates that the average failure prediction accuracy is predicted to be 0.516 if the dataset size is less than or equal to 350 *and* the failure percentage is less than or equal to 7.5. Otherwise, the failure average prediction accuracy is predicted to be 0.9.

From the regression trees, it is clear that dataset size and failure percentage are once again the two main factors that explain variations in failure prediction accuracy. Both CNN+L and CNN+B show similar results: the accuracy decreases significantly when the dataset size is less than or equal to 350 and the failure percentage is less than or equal to 7.5. BiLSTM+B also exhibits low accuracy in similar conditions (i.e., when both the dataset size and the failure percentage are small), but it additionally shows a low accuracy when the maximum log sequence length is higher than 750.

More practical implications and guidelines derived from the classification and regression trees will be further discussed in Section 2.7.1.

The answer to RQ4 is that dataset size, followed by failure percentage, plays a crucial role in the accuracy of DL-based failure predictors, while LSL is important only for certain configurations. In contrast, failure pattern type does not have a clear relationship with failure prediction accuracy.
Interestingly, failure predictors are very accurate (*F1-score* > 0.95) and robust ($IQR < 0.01$) when the dataset size is above 350 or the failure percentage is above 7.5%.

2.6.5 RQ5: Real-world Data

Table 2.9 presents the accuracy results of the synthesised datasets alongside those of the real-world dataset, OpenStack_FP, for the same best DL-based configuration, CNN+L. The "Dataset" column lists the datasets chosen for comparison. $\mathcal{D}_{\mathcal{M}_1}$, $\mathcal{D}_{\mathcal{M}_2}$, and $\mathcal{D}_{\mathcal{M}_2}$ are the datasets generated from the $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ behavioural models, with similar characteristics to OpenStack_FP in terms of dataset size, maximum log sequence length, and percentage of failure, denoted by "DS", "MLSL", and "PF", respectively. "CNN+L" stands for the most effective configuration based on RQ1-3 results. For each behavioural model, two dataset instances match these three characteristics but have different failure pattern types (*Type-F* and *Type-I*). The values of precision, recall, and F1 score (denoted by "P", "R", and "F1", respectively) are shown under the "CNN+L" column, for a more detailed comparison. Since we do not have information regarding the failure pattern types of OpenStack_FP, the table presents an average of the two synthesised datasets in each row dedicated to the Synthesised data. Furthermore, the fourth row shows the average accuracy results from all synthesised behavioural models.

According to Table 2.9, the average F1 score for synthesised datasets shows a difference below 0.01 with OpenStack_FP (0.969 vs 0.974). The precision values obtained on the synthesised datasets are slightly lower than those obtained on OpenStack_FP, while the recall values are slightly higher. The average difference amounts to 0.019 for precision and

Table 2.9: Comparison of Results from a Real-world Dataset (OpenStack_FP) with Synthesised Datasets with similar characteristics

Dataset	DS	MLSL	PF	CNN + L		
				P	R	F1
$\mathcal{D}_{\mathcal{M}_1}$	1000	500	20	0.987	1.000	0.993
$\mathcal{D}_{\mathcal{M}_2}$	1000	500	20	0.932	0.965	0.948
$\mathcal{D}_{\mathcal{M}_3}$	1000	500	20	0.947	0.988	0.967
average				0.955	0.984	0.969
OpenStack_FP	876	468	21.46	0.974	0.974	0.974

0.010 for recall. To rigorously assess the significance of this difference, we performed the Wilcoxon test between the accuracy results obtained on synthesised data and those obtained on OpenStack_FP; for each test, the accuracy results from the synthesised datasets were paired with the results from OpenStack_FP. All p-values for precision, recall, and F1 score are far above 0.05, indicating that the differences between real-world and synthesised datasets are statistically insignificant.

The answer to RQ5 is that there is no significant difference between the accuracy results obtained on comparable synthesised datasets and a real-world one (OpenStack_FP) when using the best configuration for failure prediction (CNN-based encoder with Logkey2vec).

2.6.6 Data Availability Statement

The replication package, including the implementation, generated datasets with behavioural models and results, is publicly available [88].

2.7 Discussion

2.7.1 Findings and Implications

Our study leverages the main DL types (LSTM, CNN, and transformer), along with all categories of LSE strategies (Logkey2vec, BERT, and hybrid strategy of FastText and TF-IDF). In contrast to other studies mentioned in Table 2.1, the full configuration of DL encoders and LSE strategies are evaluated. Moreover, instead of using a limited number of datasets, using synthesised data enables us to control dataset characteristics and identify the necessary conditions for achieving high-accuracy models. Nonetheless, we also considered a real-world dataset for failure prediction (OpenStack_FP) and applied it to the best failure predictor configuration. This allows us to compare the failure prediction accuracy

results obtained on the synthesised datasets with those obtained on the OpenStack FP dataset.

Several significant findings are reported in § 2.6. First, the CNN-based DL encoder performs the best among different DL encoders, including those based on LSTMs, transformers, and BiLSTMs. Second, the CNN-based DL encoder works best with the Logkey2vec embedding strategy, although BERT fares better than Logkey2vec and the hybrid of FastText and TF-IDF overall for all DL encoders. Third, compared to the leading traditional ML approaches, such as Random Forest, the best DL-based failure predictor configuration yields significantly higher accuracy and robustness. Fourth, although the CNN-based DL encoder and the Logkey2vec embedding strategy are not the most recent techniques in their respective fields, interestingly, their configuration (CNN+L) works best overall for failure prediction. For CNN+L, both the size and the failure percentage of input log datasets significantly drive the failure prediction accuracy, whereas the log sequence length and the failure pattern type do not. Similar trends have been observed for the second-best configuration (CNN+B). However, for the third-best configuration (BiLSTM+B), besides the above relations, MLSL increases the maximum length of log sequences significantly decreases the failure prediction accuracy.

Fifth, based on the analysis in Section 2.6.4, we can provide comprehensive guidelines. In general, for datasets larger than 3000, CNN+L is the recommended configuration. Conversely, when dataset sizes are 3000 or less and the dataset's failure percentage is at most 15%, the preferred choice is BiLSTM+B. Regarding the expected accuracy, the accuracy of both CNN+L and CNN+B significantly reduces when the dataset size is 350 or below, with a failure percentage of up to 7.5%. While BiLSTM+B accuracy is directly affected by the maximum log sequence length, accuracy further decreases when it exceeds 750. If the maximum log sequence is at most 750, BiLSTM+B significantly decreases in performance when the dataset size is 350 or below, similar to CNN+L and CNN+B, and the failure percentage reaches up to 15%.

The conditions driving failure prediction accuracy suggest practical guidelines. For example, for a log dataset size below 350 and a failure percentage below 7.5%, failure prediction using CNN+L will be inaccurate and cannot be trusted. In that case, one can increase either the log dataset size or the failure percentage to build a better failure predictor. Although the failure percentage is inherent to the system under analysis and may not be easily controlled in practice, collecting more log sequences during the system's operation to increase the dataset size is usually feasible. Overall, our analysis provides practical and empirically validated guidelines for selecting failure prediction configurations within the studied range of dataset characteristics. These guidelines are derived from comprehensive evaluations across diverse datasets and configurations. However, for datasets with characteristics outside these ranges, additional empirical research will be necessary to determine the extent to which our guidelines remain valid and to refine these guidelines accordingly.

Last but not least, using the best configuration, the accuracy results obtained on synthesised and real-world datasets do not present a significant difference, hence further suggesting our data synthesis approach is valid.

Below, we discuss the practical implications of our findings for the main stakeholders:

AIOps engineers and software engineering researchers.

AIOps Engineers. Proactive maintenance is an important part of AIOps engineering [89]. Failure prediction is, therefore, a crucial part of alleviating the impact of failures. In this study, the analysis of the best configurations of the failure prediction model described in § 2.6.2 can guide engineers in choosing the most appropriate options when designing an architecture for their data. Our guidelines, based on the decision and regression trees presented in § 2.6.3, narrow the scope of possible design choices by decreasing the number of candidate configurations based on the characteristics of the dataset. Furthermore, we remark that the implementation of our modular architecture is available (see § 2.6.6), enabling AIOps engineers to reuse our artifacts seamlessly.

Software Engineering Researchers. In this chapter, we use a modular architecture to effectively study different DL architectures on failure prediction data. Since existing approaches apply DL models with selective settings such as LSE strategies [6,7], we propose a novel approach to study configurations of LSE strategies and DL architectures that have not been studied together before (see Table 2.1). We speculate this approach can further inspire the adaptation of DL-based modular architectures in other studies in the field of AI for software engineering. In addition, we use a controllable synthetic data generation algorithm to generate labeled datasets with varying characteristics. Such datasets are crucial to obtaining comprehensive and generalisable results when only a limited number of datasets are available for assessing a new method. We believe the algorithm presented in § 2.5.3 can be adopted to generate synthetic datasets tailored to specific requirements.

2.7.2 Threats to Validity

There are a number of potential threats to the validity of our experimental results.

Hyperparameter tuning of models. The hyperparameters of failure predictors, such as optimizers, loss functions, and learning rates, can affect the results. To mitigate this, we followed recommendations from the literature. For the batch size and the number of epochs, as mentioned in § 2.5.2.4, we chose values for different combinations of dataset characteristics based on preliminary evaluation results. Better results could be obtained with different choices.

Synthetic data generation process. Due to the lack of a method to generate the datasets satisfying different dataset characteristics mentioned in § 2.5.3.1, we proposed a new approach, with precise algorithms, that can generate datasets in a controlled, unbiased manner as discussed in § 2.5.3. To mitigate any risks related to synthetic generation, we provided proof of the correctness of the algorithms and explained why it is unbiased during the generation process in § 2.5.3.6. To further support the validity of the generation process, in § 2.6.5, we compared the results on actual datasets reported in the literature with those

of the synthesised datasets for corresponding key parameters (e.g., dataset sizes and failure percentage). Results show to be remarkably consistent, thus backing up the validity of our experiments.

Timeliness of failure predictions. Depending on the context, the timeliness of failure prediction may impact the applicability of our DL models. Because the focus of our experiments is on prediction accuracy, we have not investigated how early our DL models can accurately predict failures; we simply predict failures after processing all log messages (up to the moment before the failure message occurs) within a log sequence. Investigating timeliness would require entirely different experiments; for example, this can be done by varying the distance between the last log message inputted to DL models and the occurrence of failures, either in terms of the number of log messages or time difference. However, due to the objective and design of our study, we use the entire log sequence before the failure for prediction, meaning the distance between the last log message in the observation window inputted to DL models and the failure log message is zero *by design*. We remark that for the real-world OpenStack_FP dataset, which contains timestamps, the average time distance between the last message before failure and the failure message is 1.87 s. However, interpreting whether such a lapse is sufficient in practice requires knowing the practical context in which the prediction models are deployed. We acknowledge the limitations of our datasets and the need to study the timeliness of failure prediction for DL models systematically in the future.

Behavioural models and failure patterns. The behavioural models and failure patterns used for generating synthetic datasets may have a significant impact on the experimental results. We would like to note that this is the first attempt to characterise failure patterns for investigating failure prediction performance. To mitigate this issue, we carefully chose them based on pre-defined criteria described in § 2.5.4 and provided a remark on its generalizability in § 2.5.4.3. Nevertheless, more case studies, especially considering finer-grained failure patterns, are required to increase the generalizability of our findings and implications and, for that purpose, we provide in our replication package all the artifacts required.

Possible bugs in the implementation. The implementation of the DL encoders, log sequence embedding strategies, dataset generation algorithms, and scripts used in our experiments may contain unexpected bugs. To mitigate this risk, we used the replication packages of existing studies [37, 43] as much as possible. Additionally, we conducted thorough code reviews.

2.8 Conclusion

In this chapter, we presented a comprehensive and systematic evaluation of alternative failure prediction strategies relying on DL encoders and log sequence embedding strategies. We

presented a generic, modular architecture for failure prediction that can be configured with specific DL encoders and embedding strategies, resulting in different failure predictors. We considered Logkey2vec, BERT, and a hybrid of FastText and TF-IDF, representing three categories of log sequence embedding strategies. We also covered the main DL categories resulting in four DL encoders (LSTM-, BiLSTM-, CNN-, and transformer-based). Our selection was inspired by the previously used DL models in the literature.

We evaluated the failure prediction models on diverse synthetic datasets using three behavioural models inferred from available system logs. Four dataset characteristics were controlled when generating datasets: dataset size, failure percentage, Log Sequence Length (LSL), and failure pattern type. Using these characteristics, 360 datasets were generated for each of the three behavioural models.

Evaluation results show that the accuracy of the CNN-based encoder is significantly higher than that of the other encoders, regardless of dataset characteristics and embedding strategies. Among the three embedding strategies, pretrained BERT outperformed Logkey2vec and the hybrid strategy overall, although Logkey2vec fared better for the CNN-based encoder. Compared to the best traditional ML-based failure predictor (Random Forest), the best configuration demonstrates significantly superior accuracy and robustness. The analysis of dataset characteristics confirms that increasing the dataset size and the failure percentage both improve failure prediction accuracy. In comparison, LSL is a significant factor only for specific configurations, while the other factors (i.e., failure pattern type) did not show a clear relationship with accuracy. Furthermore, the accuracy of the best configuration (i.e., CNN-based with Logkey2vec) consistently yielded high accuracy when the dataset size was above 350 *or* the failure percentage was above 7.5%, which makes it widely usable in practice. Finally, the accuracy results obtained from the synthesised and real datasets are consistent.

As part of future work, we plan to further evaluate the best-performing configurations of the failure prediction architecture on additional real-world log data to further investigate the effect of other factors, such as log parsing techniques or data noise, on model accuracy. As a future research direction, we plan to assess the impact of more dataset characteristics on log-based failure prediction. This notably includes different sources of data noise, such as varying degrees of mislabelled logs, log parsing errors, and evolving logs. The degree and type of data noise are, however, dependent on the system of study, and such noise may not be significant on all datasets. Finally, following the discussion on the timeliness of failure prediction and limitations of our datasets in § 2.7.2, when using real-world data, we also plan to include additional evaluation metrics, such as lead time [90] and the number of log messages before the occurrence of a failure, to assess the accuracy of models at predicting failures early on.

Acknowledgements. This work was supported by the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC), by a University of Luxembourg's joint research program grant, the Science Foundation Ireland under Grant 13/RC/2094-2, and by European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254 (COSMOS). The experiments conducted in this work were enabled in part by Digital Alliance of Canada

(alliancecan.ca).

Chapter 3

Data-efficient Anomaly Detection on Unstable Logs using ML and LLM

3.1 Overview

As discussed in Chapter 1, various software-intensive systems, such as online service systems and Big Data systems, have permeated every aspect of people's daily lives. As the prevalence of such systems continues to grow, the potential impact of software failures has become increasingly significant. A critical software failure can result in service interruptions, financial losses, and, in severe cases, pose threats to human safety [91].

Log-based anomaly detection has emerged as a promising approach to enhancing the dependability of software-intensive systems. An anomaly detector aims to discern anomalous patterns within system logs, which serve as vital indicators of the system's operational state. Early research predominantly employed classical machine learning techniques, such as Principal Component Analysis [8], Isolation Forest [9], and one-class SVM [92] for automated anomaly detection. However, these methods overlook the contextual information of the logs and, as a result, exhibit less effectiveness on more challenging cases [13, 14, 37]. In recent years, Deep Learning (DL)-based methods have gained significant traction in anomaly detection. Unlike classical machine learning methods, DL methods typically consist of a large number of trainable parameters, enabling them to model long contextual dependencies and complex semantic patterns in logs. In particular, log-based anomaly detection has significantly benefited from sequential DL models such as LSTM and transformers, achieving high predictive performance on multiple benchmark datasets [10, 11, 16, 93]. Despite the success of DL-based methods, we highlight three key challenges prevalent in current practices of log-based anomaly detection:

C1 Existing approaches often assume a stable data distribution, which is unrealistic in real-world scenarios. In practice, unlike current benchmark datasets, where log structures and contents remain stable, logs can be *unstable* due to software evolution or environment changes. The majority of anomaly detection methods [10, 11, 16, 22, 37, 51, 94, 95] have been proposed for and evaluated on stable log

datasets. In contrast, only a few studies [13, 14, 59] have investigated anomaly detection on unstable logs, mainly due to a lack of public benchmarks. Earlier works leverage private or synthetic data. However, more recently, Huo et al. [13] have proposed two public datasets for unstable logs.

C2 Machine Learning-based (ML) anomaly detection, especially when based on DL methods, often relies on substantial labeled data, which is costly to obtain. The most effective methods in anomaly detection—particularly those based on DL—often rely on an extensive amount of labeled data for their training, due to their substantial number of trainable parameters. Collecting such data requires intensive labor and significant domain knowledge in practice. More recently, the study of Yu et al. [96] has demonstrated that simpler methods, such as Decision Trees (DT), exhibit greater effectiveness. However, they only evaluated their methods on stable logs.

C3 The reported effectiveness of ML-based anomaly detection might be inflated due to data leakage issues. Yu et al. [12] found such issues in several benchmark datasets such as HDFS [97] and BGL [98], where the testing data contains training instances. This leakage can potentially boost the effectiveness of ML methods, especially DL methods, as their large parameter set allows them to memorize the training data. To address this issue, Yu et al. [96] removed instances in testing data that were already seen in the training data, which led to a significant drop in anomaly detection effectiveness.

In light of these challenges, we identify the next frontier of log-based anomaly detection as addressing a more realistic and demanding task: *anomaly detection on unstable logs with limited labeled data (ULAD)*. Unlike *anomaly detection on stable logs (SLAD)*, ULAD reflects the practical reality where logs evolve due to software updates or environmental factors, resulting in instability (C1). This evolution results in changes such as the addition, removal, or modification of log messages, as well as shifts in their order. Furthermore, real-world constraints often limit the availability of labeled data, which is costly to collect and requires domain knowledge (C2). To ensure a realistic assessment of ULAD solutions, test instances already seen in the training data should be removed from testing data, addressing issues of data leakage (C3).

The literature on anomaly detection in the presence of unseen log templates [10, 16, 51] is related to the ULAD challenge. However, ULAD is more challenging since, although unstable logs are test instances not present in training data, they further follow a different log distribution than historical data. Moreover, these works do not address the C2 and C3 challenges. Although unsupervised methods have been explored to reduce the reliance on labeled data, they remain insufficient in practice due to their limited effectiveness under log evolution and instability (C1), where log structures and contents change over time [13, 14]. Additionally, the absence of supervision makes it difficult to capture subtle anomalies when training data is scarce (C2). Last, these methods once again overlook the challenge of data leakage (C3).

A promising approach to tackling these challenges of ULAD lies in leveraging Large Language Models (LLMs). Recently, LLMs have gained significant attention for their ability to mitigate the data insufficiency problem faced by ML-based methods. By pretraining

on vast, diverse datasets, LLMs can excel in tasks with limited labeled data. Several researchers have investigated various prompts to instruct pretrained LLMs such as GPT to perform anomaly detection directly (i.e., in-context learning) [99–101]. An alternative to in-context learning is fine-tuning, where extra training on domain-specific data is required. While in-context learning has been more widely studied because it does not require additional training and can be applied directly with prompts, Mosbach et al. [102] highlighted its poor generalizability on challenging tasks. Drawing from this observation, fine-tuning may be a more suitable strategy for ULAD when using LLMs.

Most recent works [99, 101, 103–106] in log analysis focused on using closed-source LLMs from OpenAI, due to their user-friendly environment and effective performance. However, these LLMs induce a significant financial cost and show unpredictable latency during training and inference [107]. On the other hand, open-source LLMs are free to use, and we have some degree of control in terms of fine-tuning algorithms and inference time.

Though a fine-tuned LLM can address challenges C1, C2, and C3 faced by ML methods, they are inherently designed for textual understanding and generation rather than the detection of anomalous patterns in logs. Conversely, existing anomaly detectors using ML models such as Decision Tree (DT) and Single-layer Feedforward Network (SLFN) have proven to be effective in the SLAD task when abundant data is available for training [12], demonstrating their capacity to detect anomalous patterns. This indicates that combining ML methods with an LLM would leverage the strengths of both approaches, ML models' ability to detect anomalous patterns and fine-tuned LLM's capacity to handle scarce labeled data effectively.

To this end, we propose *FlexLog*, a novel approach that requires significantly less labeled data for ULAD compared to ML methods. FlexLog integrates ML-based anomaly detectors and an LLM, combining their strengths to enhance effectiveness and data efficiency. We summarize our contributions as follows.

- **Dataset configuration for ULAD.** Most existing benchmark datasets contain stable logs, used for the SLAD task. In this chapter, we selected three of these datasets—HDFS, LOGEVOL, and ADFA-LD (referred to as ADFA for brevity hereafter)—and configured four unstable datasets for ULAD, namely SynHDFS-U, LOGEVOL-U, SYNEVOL-U, and ADFA-U, by deliberately introducing disparities between the training and testing datasets. To eliminate the influence of data leakage (C3) [12] and further increase instability, we performed deduplication on each dataset, ensuring that any data samples included in the testing datasets were excluded from the training datasets.

- **FlexLog, a novel approach for ULAD equipped with practical strategies to boost effectiveness and efficiency.** FlexLog uses average-based ensemble learning to combine the predictive strengths of a fine-tuned LLM and ML methods, capturing intricate anomalous patterns with only limited labeled data for training. Specifically, to address C2, FlexLog employs Parameter-Efficient Fine-Tuning (PEFT) of a pretrained LLM, leveraging its vast embedded knowledge to mitigate

the constraints of scarce labeled data. To tackle C1, FlexLog integrates Retrieval-Augmented Generation (RAG) to dynamically incorporate external knowledge and enhance the model's adaptability to unstable log distributions. Additionally, a cache mechanism improves computational efficiency by eliminating redundant operations.

- **State-of-the-art effectiveness and data efficiency for ULAD.** To evaluate FlexLog, we first compare it against baselines trained on full datasets, even though FlexLog itself is trained on significantly smaller datasets. This comparison is conducted on two real-world datasets (ADFA-U and LOGEVOL-U) and two synthesized datasets (SynHDFS-U and SYNEVOL-U). Experimental results show that FlexLog achieves state-of-the-art effectiveness, outperforming the top baseline by at least 1.2 percentage points (pp) in terms of F1 score, while reducing the usage of labeled data by more than 62.87 pp. Further, we assess the data efficiency of FlexLog by comparing it with baselines when trained on the same datasets. Experiment results on ADFA-U show that FlexLog consistently outperforms all baselines across varying training dataset sizes, except in the extreme data scarcity scenario (where the training dataset size is 50), where all methods exhibit poor performance due to insufficient labeled data. FlexLog achieves a maximum gain of 13 pp in F1 score when the training dataset size is 500. This confirms FlexLog is the most effective choice when only limited labeled data is available.

The rest of the chapter is organized as follows. Section 3.2 presents the basic definitions and concepts that will be used throughout the chapter. Section 3.3 describes our data-efficient anomaly detection approach, FlexLog. Section 3.4 presents our experimental design. Section 3.5 outlines our results, discusses the implications, and describes the threats to the validity of our study. Section 3.6 presents related works and finally, Section 3.7 concludes and suggests future directions for research and improvement.

3.2 Background

3.2.1 Anomaly Detection on Logs

Anomalies in logs refer to logs that do not conform to the normal behavior of a system [108]. Log-based anomaly detection represents a binary classification task to identify anomalies from logs. Depending on their distributions, logs can be divided into two categories: *stable logs* (Definition 1) and *unstable logs* (Definition 2).

Definition 1 (Stable Logs)**.** Logs drawn from a single underlying distribution, i.e., their structure and semantics remain consistent in all the logs.

Definition 2 (Unstable Logs)**.** Logs drawn from more than one underlying distribution.

Stable logs are typically generated from systems whose logging behaviors and operating environment remain unchanged over time, resulting in consistent structure and semantics

of logs. In contrast, unstable logs originate from multiple distributions caused by system or environmental changes. *System evolution* refers to internal changes within a software system, such as version upgrades. Developers often modify source code, including the addition of logging statements, which can result in changes to the logs. As Yu et al. [109] reported, around 24 %–40 % of log statements change during their lifetime. Taking the public dataset LOGEVOL as an example, 24 % of logging statements were modified during the system upgrade from Spark version 2 to 3 [13]. As a result, 14 % of logs collected in Spark 3 contain new log templates induced by system evolution. This figure represents a conservative estimate of the percentage of unstable logs, as other causes of instability (e.g., reordering of logging statements during execution) are not accounted for due to the lack of a mapping between execution paths and their log distributions. Nonetheless, these results clearly highlight that unstable logs are common in practice, underscoring the importance of handling instability caused by system evolution. *Environmental evolution*, on the other hand, represents the changes of external factors, such as a shift of user distribution and the emergence of unseen attack types. These changes affect both normal and abnormal patterns in the logs by altering the structure, content, or frequency of log messages. For example, shifts in user distributions—such as changes in user geographic regions—may introduce new log sequences or alter the frequency of existing ones due to differences in usage patterns, device configurations, or regional preferences. Similarly, novel or previously unseen attacks may generate anomalous logs with sequences or templates that have not been observed in the earlier log distribution.

Built on the definitions of stable and unstable logs, we define two corresponding anomaly detection tasks, namely *Anomaly Detection on Stable Logs (SLAD)* and *Anomaly Detection on Unstable Logs (ULAD)* as defined in Definition 3 and 4, respectively.

Definition 3 (Anomaly Detection on Stable Logs)**.** SLAD is a binary classification task that aims to predict anomalies in stable logs, i.e., the training data and testing data follow the same distribution.

Definition 4 (Anomaly Detection on Unstable Logs)**.** ULAD is a binary classification task that aims to predict anomalies in unstable logs, i.e., the training data and testing data are drawn from different distributions.

SLAD is the predominant configuration in the literature [14, 110], largely because existing benchmark datasets often assume a stable software system. In this configuration, the training dataset $\boldsymbol{D}^{train} = \{ls_1, ls_2, ..., ls_n\}$ and the testing dataset $\boldsymbol{D}_S^{test} = \{ls_{n+1}, ls_{n+2}, ..., ls_{n+m}\}$ are sampled from the same distribution. Despite its wide adoption, SLAD does not reflect challenges faced by real-world applications, e.g., evolving systems or operating environments. In contrast, ULAD (Definition 4) considers a more challenging yet realistic scenario. Specifically, the training dataset $\boldsymbol{D}^{train}$ consists of stable logs collected under consistent conditions, while the test dataset $\boldsymbol{D}_U^{test}$ contains unstable logs resulting from system or environmental evolution.

Evaluation on Deduplicated Datasets. As demonstrated in previous work [12], data leakage is prevalent in existing benchmark datasets, artificially inflating the effectiveness of

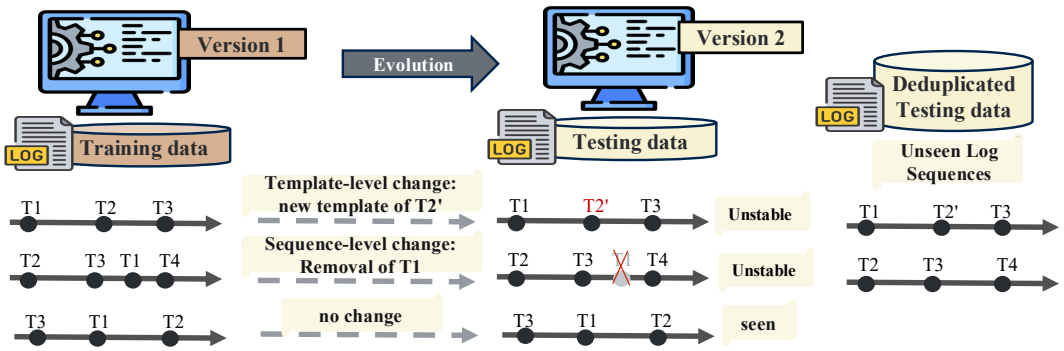


Figure 3.1: Examples of Unstable Logs Resulting from Log Evolution.

anomaly detectors. Data leakage entails an overlap between testing and training data, i.e., some log sequences in the testing dataset have already been seen in the training dataset. This phenomenon affects both the SLAD and ULAD tasks. To eliminate the risk of data leakage, we remove *seen* testing log sequences that are already present in the training dataset D^{train}, yielding a new testing dataset for ULAD and SLAD, denoted as $D^{test}_{U\dagger}$ and $D^{test}_{S\dagger}$ respectively, defined in Equations 3.1 and 3.2.

$$D^{test}_{U\dagger} = D^{test}_{U} \setminus D^{train} \tag{3.1}$$

$$D^{test}_{S\dagger} = D^{test}_{S} \setminus D^{train} \tag{3.2}$$

$D^{test}_{U\dagger}$ and $D^{test}_{S\dagger}$ consist of only *unseen log sequences*. These sequences differ from the ones in the training dataset at two possible levels: 1) template level (when there is an *unseen log template* in the sequence), 2) sequence level (when all the log templates are already mentioned in the training data but their order is new). Unseen log sequences can be either stable or unstable, depending on their underlying log distributions. As mentioned by Yu et al. [96], after deduplication, the effectiveness of anomaly detection models on testing data drops, making it a more challenging task in this realistic scenario. We note that Yu et al. [12] applied deduplication to the entire dataset prior to splitting it into training and testing sets, whereas we performed deduplication after the split. This ensures the removal of information leakage with minimal changes to the original data distribution, since having replication within training or within testing data is both realistic and common in practice.

Illustrative Examples. Figure 3.1 provides an overview of the deduplicated ULAD with example log sequences. After system evolution from version 1 to version 2, the first two sequences at the top undergo changes at different levels. In the first sequence, $T1 \rightarrow T2 \rightarrow T3$, template $T2$ is updated to a new template $T2'$, representing a change at the template level. The second sequence, $T2 \rightarrow T3 \rightarrow T1 \rightarrow T4$, experiences a change at the sequence level, where template $T1$ is no longer present. The last sequence shown, $T3 \rightarrow T1 \rightarrow T2$, is regenerated in the later version without any change, and is therefore marked as a "seen" sequence relative to the training data. During deduplication, the seen

sequence is removed from the test set. The remaining sequences are referred to as unseen log sequences, as they contain no overlapping sequences.

3.2.2 Task Adaptation Strategies for Large Language Models

LLMs typically consist of substantial parameters pretrained on vast and diverse datasets, possessing knowledge across various domains. However, how to effectively adapt pretrained LLMs to domain-specific tasks remains an open problem. Two predominant strategies for task adaptation are In-context Learning (ICL) and Fine-Tuning (FT).

ICL operates without altering the weights of the LLMs [111]. Instead, it leverages prompts—structured textual inputs—to guide the model's behavior. These prompts typically include task instructions and, in some cases, a series of demonstrations in a conversation between the user and the assistant. In a classification task, e.g., ULAD, each demonstration consists of an input x paired with its corresponding ground-truth label y. When no demonstrations are provided, the approach is referred to as zero-shot ICL, whereas the inclusion of a few demonstrations constitutes a few-shot ICL.

Although ICL is relatively easy to implement, it faces several challenges and limitations, including issues with efficiency, scalability, generalizability, and high financial cost when using closed-source LLMs [112]. As an alternative, FT alleviates these issues by training pre-trained LLMs with domain-specific data. In practice, there are two main types of fine-tuning, namely API-based FT and Custom FT.

API-based FT refers to fine-tuning performed through dedicated APIs made available by the LLM provider, e.g., OpenAI [113]. This is typically the case for closed-source LLMs, such as GPT-3.5 [114] and GPT-4 [115], for which neither full nor selective fine-tuning is allowed without accessing their APIs. These APIs support fine-tuning a set of prompt-completion pairs or conversations, depending on whether LLMs are used in purely generative or conversational settings.

Custom FT, on the other hand, applies to open-source LLMs, such as LLama [116] and Mistral [117]. Common custom FT techniques include full fine-tuning and Parameter-Efficient Fine-Tuning (PEFT). Let the trainable parameter set of an LLM be denoted as $\boldsymbol{W}$, the task-specific dataset as $\boldsymbol{D}$, and its associated label set as $\boldsymbol{L}$.

- *Full Fine-Tuning*: This approach utilizes gradient descent-based optimizer to update $\boldsymbol{W}$ to $\boldsymbol{W}^f$, thereby adapting the LLM to a specific task. Specifically, prompts are constructed using $\boldsymbol{D}$ and fed into an LLM. The model's output distribution $\hat{y}$ is then compared with the corresponding label distribution y, using a distribution-level loss, e.g., cross-entropy loss. This loss guides weight updates from $\boldsymbol{W}$ to $\boldsymbol{W}^f$ through backpropagation.

- *Parameter-Efficient Fine-Tuning:* PEFT preserves original LLM weights while training only a small number of task-specific adapter layers and parameters. There are several types of PEFT methods, including additive, selective, reparameterized, and hybrid PEFT [118]. The predominant PEFT techniques are LoRa [119] and its

derivative techniques such as QLoRa [120]. Essentially, LoRa uses a low-rank decomposition to reduce computational cost while maintaining performance similar to full fine-tuning as in Equation 3.3.

$$\begin{aligned} W^f &= W + \Delta W \\ &= W + AB \end{aligned} \tag{3.3}$$

where $A \in \mathbb{R}$ and $B \in \mathbb{R}$ are lower rank matrices compared to W, with dramatically fewer trainable parameters. Such techniques significantly reduce the computational cost while maintaining comparable performance to fully fine-tuned LLMs.

3.3 Methodology

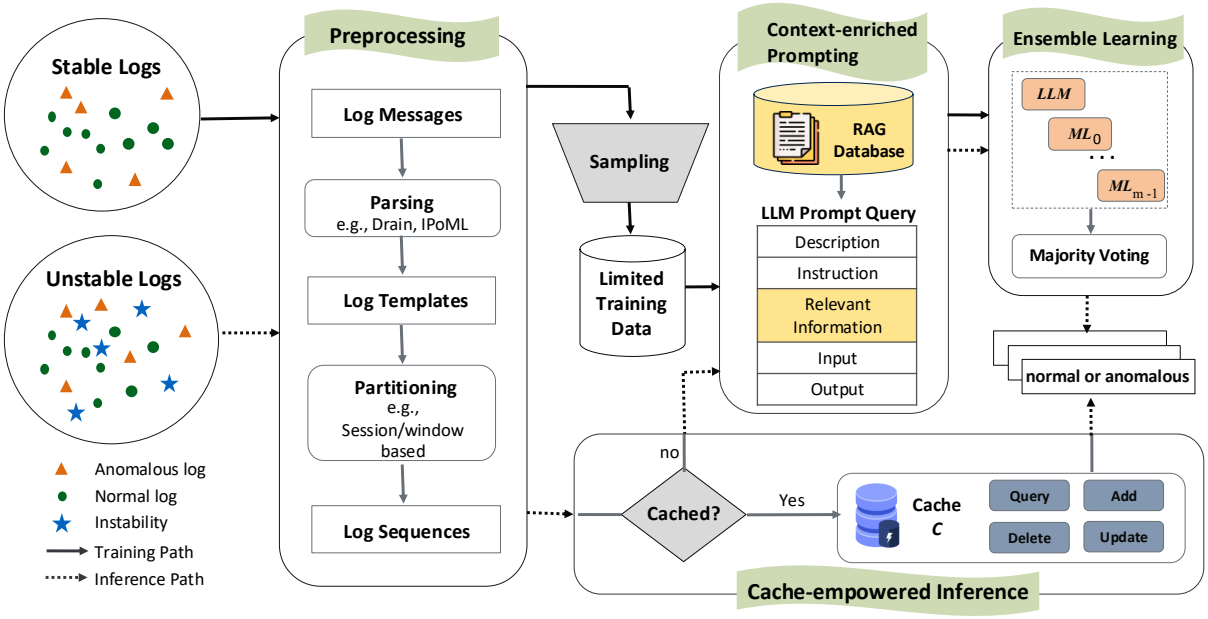


Figure 3.2: Architecture of FlexLog.

In this chapter, we propose a novel approach, namely FlexLog, to tackle ULAD by synergizing the capabilities of ML methods and LLMs via ensemble learning. Specifically, FlexLog combines the predictions from trained ML methods and a fine-tuned LLM to make a final decision. This approach leverages the strengths of both paradigms: ML methods excel at capturing anomalous patterns within logs, while LLM brings broad prior knowledge from pretraining, allowing them to adapt to novel log patterns even with limited labeled data. Also, we tackle the three key challenges in ULAD— unstable log distribution (C1), data insufficiency (C2), and data leakage (C3) — by employing ensemble learning of ML and LLM, PEFT, and deduplication in the testing data, respectively. Additionally, we further tackle unstable log distributions (C1) by using RAG in prompting when relevant external information is available regarding log sequences.

Figure 3.2 illustrates the architecture of FlexLog, which comprises four main components: preprocessing (§ 3.3.1), cache-empowered inference (§ 3.3.2), context-enriched prompting (§ 3.3.3), and ensemble learning (§ 3.3.4). FlexLog is designed to predict whether unstable logs—generated by software systems that have undergone software or environmental evolution—are anomalous. Specifically, the *preprocessing component* converts raw stable and unstable logs into log sequences by extracting log templates and grouping them into log sequences using either window-based or session-based partitioning. For illustrative purposes, consider a log sequence ls_i as an example. The *cache-empowered inference component* first checks if ls_i matches an existing entry in the cache. If a match is found, FlexLog retrieves the stored prediction as the output label directly. Otherwise, the *context-enriched prompting component* uses ls_i in a structured prompt enriched with contextual information, such as log event descriptions and Linux system call names. This prompt includes key fields such as description, instructions, relevant information (optional), input, and output. It serves as input both for fine-tuning of and for inference with an LLM, which acts as one of FlexLog's base models. To construct the ensemble, the *ensemble learning component* fine-tunes the LLM-based model (e.g., Mistral or Llama) and trains the ML-based models (e.g., DT and KNN [96]) on a limited dataset sampled from the preprocessed stable logs to maintain data efficiency and reduce training overhead. During inference, if no matching log sequence is found in the cache, binary anomaly predictions from these base models are aggregated using majority voting to produce the final decision. This prediction is then stored in the cache to optimize future queries. In the following sections, we provide a detailed explanation of each component.

3.3.1 Preprocessing

As illustrated in the first box of Figure 3.2, the *preprocessing component* transforms raw log messages to log sequences via two primary processes, namely parsing and partitioning. Given a raw log message (e.g., "`12:03 INFO Sent Block 12`"), we leverage a log parser (e.g., Drain [81]) to identify the static parts (e.g., "`Sent Block`") and dynamic parts (e.g., "`12`") and replace the latter with the symbol `"<*>"`.

The parsing process yields log templates, which are then fed into the partitioning process. This process aggregates log templates into log sequences based on their session IDs or a fixed-size window (as described in § 2.2.2). In general, session-based partitioning is favored, as long as the maximum sequence length does not exceed the input limit of the models being used, since it is reported to yield better results than window-based partitioning for anomaly detection [110].

3.3.2 Cache-empowered Inference

FlexLog maintains a cache C as illustrated at the bottom of Figure 3.2. C stores previously seen log sequences along with their predicted labels. Given a log sequence ls_i under detection, FlexLog first queries cache C for matching entries. If an identical log sequence is found, the corresponding label l_i is retrieved and used directly, bypassing the need for

additional computation by RAG and ensemble learning. Conversely, if no match is found in C, FlexLog performs RAG-based prompting (§ 3.3.3) and leverages ensemble learning (§ 3.3.4) to predict the label for ls_i and subsequently adds the new log sequence and its label to the cache.

The maintenance of C involves four core functions, namely *query*, *add*, *update*, and *delete*. The *query* function compares the input log sequence against the entries in C and returns the identical entry along with its associated label if a match is found. The *add* function inserts a new log sequence and its predicted label into C if it is not present. The *delete* function removes a specific entry from C, allowing FlexLog to manage cache size or discard outdated information. The *update* function modifies an existing entry's label, incorporating human corrections to improve future predictions.

3.3.3 Context-enriched Prompting

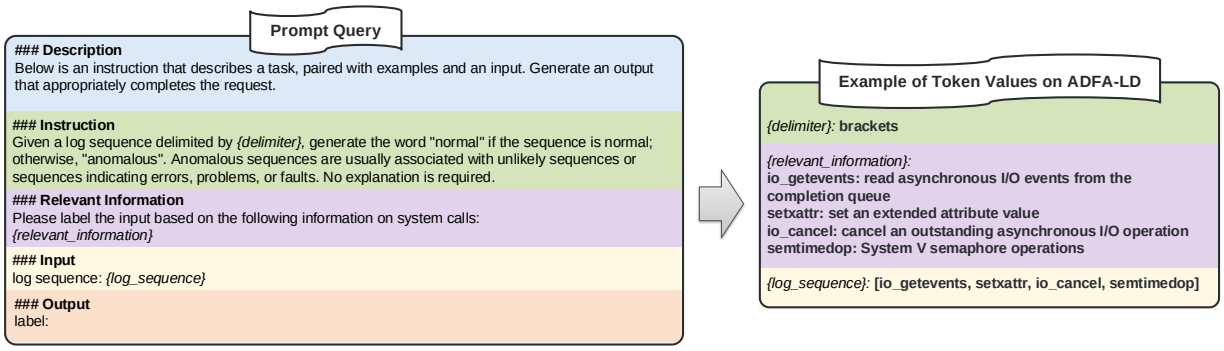


Figure 3.3: FlexLog's Prompt Design for LLM Fine-tuning and Inference.

This component processes the log sequences that are not stored in the cache, preparing them for the LLMs used in FlexLog. Log sequences are inherently challenging for LLMs to understand because these models are primarily designed for Natural Language Processing (NLP) tasks and are better suited to processing and reasoning over textual data. To bridge this gap, we place log sequences with their log templates into semi-structured prompts that resemble natural language, enabling LLMs to leverage their NLP capabilities effectively. The RAG component enhances these prompts with additional external information. Specifically, for each unique template, we maintain a RAG database containing associated descriptions (e.g., system call explanations). When constructing a prompt, the mechanism searches this database for the templates present in the input log sequence and retrieves the relevant context, which is then inserted into the prompt. In this way, the RAG mechanism enriches the input provided to the LLM during fine-tuning and inference.

To devise an effective prompt structure for ULAD, we adopt the tactics reported by Winteringham [121]. Concretely, as illustrated in Figure 3.3, each prompt comprises five parts, namely, description, instruction, relevant information, input, and output. Notably, the retrieved context is included only when pertinent information is available. We provide the details of each part next.

Description. This part sets the overall context of the task for the LLM. It provides a

high-level overview of what the model is expected to accomplish. For instance, as shown in Figure 3.3, the description explains that the LLM should generate an output that appropriately completes the task request. While this part does not specify the input or output format explicitly, it prepares the LLM by providing a concise summary of the task objective.

Instruction. The instruction part formally introduces the task by describing its goal, input format, and output expectations. It specifies how the LLM should process the input and produce the desired label. After experimenting with multiple instruction formats, we present the most effective formulation in Figure 3.3. In this part, we describe the delimiter for log sequences (i.e., "brackets") and add "No explanation is required", instructing the LLM to output only the label.

Relevant Information (Optional). This part includes additional information related to the log sequence, which enhances the LLM's ability to interpret the input. When available, contextual data is retrieved and presented in this part to provide background information. As described above, this information is retrieved from the RAG database by searching for the templates contained in the log sequence and returning their stored descriptions. For example, the right-hand side of Figure 3.3 includes system call descriptions, such as `setxattr` or `semtimedop`, which help clarify the function and purpose of the operations within the log sequence. Including such information enables the LLM to better understand the relationships between the components of the log sequence, thereby improving its ability to generate accurate predictions. However, if no relevant external context is available, this part of the prompt is omitted.

Input. This part presents the log sequence to be analyzed in the format *"log sequence: {log_sequence}"*, where *{log_sequence}* is a placeholder dynamically replaced with different log sequences during fine-tuning and inference. For example, the right-hand side of Figure 3.3 shows the replacement of the placeholder with the log sequence *"[io_getevents, setxattr, io_cancel, semtimedop]"* from the ADFA-U dataset.

Output. The output section guides an LLM in predicting the label of the input log sequence. It provides a formatted prompt that ends with "label:", prompting the LLM to generate the next token as either "normal" or "anomalous", based on its analysis of the log sequence.

3.3.4 Ensemble Learning

The goal of this component is to maximize the utility of limited labeled data by integrating different base models, each offering a unique perspective on performing ULAD effectively. To train the base models, we leverage the stable logs collected from software systems before undergoing the software or environment evolution. A salient feature of FlexLog is employing both ML- and LLM-based models as base models as introduced in § 3.1. Due to LLMs' pretraining on diverse corpora, they can be effectively fine-tuned with only limited data. Consequently, we sample only a subset from the stable logs to create the training dataset. Using this training dataset, FlexLog fine-tunes an LLM and fits m ML models $\{ML_0, \ldots, ML_{m-1}\}$.

For a given LLM, FlexLog adopts (see Equation 3.4) API-based fine-tuning for closed-

source LLMs (e.g., GPT 4o) and LoRa for open-source LLMs (e.g., Llama and Mistral); details about API-based fine-tuning and LoRa are provided in Section 3.2.2. We denote the fine-tuned LLM as LLM^f.

$$LLM^f = \begin{cases} API\text{-}based\_FT(LLM, data = S_{train}, label = L) & LLM \in \text{Closed-source LLMs} \\ LoRa(LLM, data = S_{train}, label = L) & LLM \in \text{Open Source LLMs} \end{cases} \tag{3.4}$$

For a given ML model ML, gradient descent optimization is applied for neural network-based models, while model-specific fitting methods are used for non-parametric models such as DT (Equation 3.5). Note that K-Nearest Neighbors (KNN) does not involve a traditional training or fitting process but instead relies on distance-based comparison during inference. The resulting learned ML model is denoted as ML^f.

$$ML^f = \begin{cases} gradient\_descent(ML, data = S_{train}, label = L) & ML \in \text{Neural Networks} \\ fit\_dt(ML, data = S_{train}, label = L) & ML = \text{DT} \\ distance\_based\_comparison & ML = \text{KNN} \end{cases} \tag{3.5}$$

After training individual models, the next step is to combine their outputs using *majority voting*, a commonly used, simple yet effective ensemble learning technique in the literature [122–124]. Formally, let $\mathcal{M} = \{M_0, M_2, \ldots, M_{N-1}\}$ be the set of N learned base models, with $\mathcal{M}_i \in \mathcal{M}$ representing either a learned LLM LLM^f or an ML-based method ML^f. For a given log sequence ls_i, each base model $\mathcal{M}_i$ predicts the label y_i of x, equals 1 if anomalous, and 0 if normal. The final label is determined by a majority voting function $MV(\cdot)$ among all base models, as shown in Equation 3.6.

$$\phi_i = MV(ls_i) = \begin{cases} 1, & \text{if } \sum_{i=0}^{N-1} y_i > \frac{N}{2} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

Here, ϕ_i represents the final prediction for ls_i. In case of a tie (when exactly half of the votes are normal), the sequence is classified as *normal*, following our assumption that anomalies are rare.

3.4 Experimental Design

3.4.1 Research Questions

We investigate the following research questions:

RQ1 (**effectiveness**) How effective is FlexLog for ULAD compared to the baselines?
- **RQ1.1** Can FlexLog trained on limited labeled data achieve comparable effectiveness to baselines trained on full datasets?

RQ1.2 What impact does the level of log instability have on FlexLog and the baselines?

RQ2 (**data efficiency**) How does the amount of labeled training data impact FlexLog's effectiveness, and can it maintain robust effectiveness under varying degrees of data scarcity?

RQ3 (**time and memory efficiency**) What is the performance of FlexLog in terms of time efficiency during training and inference, and how much memory overhead does the cache incur during inference?

RQ4 (**configuration impact**) How does the performance of FlexLog vary under different configurations, including ablations of base models, RAG, and the cache, as well as alternative LLM choices?

RQ1 investigates the overall effectiveness of FlexLog for the ULAD, comprising two sub-RQs. With RQ1.1, we aim to demonstrate the strengths of FlexLog when only limited labeled data is available. Specifically, we compare baselines trained with full datasets against FlexLog trained with much smaller subsets. With RQ1.2, we aim to highlight the distinct advantage of FlexLog in handling gradually increasing instabilities in ULAD. To this end, we assess the effectiveness of FlexLog and baselines on SLAD and under the influence of varying ratios of instability in both log templates and sequence levels. RQ2 focuses on data efficiency by training FlexLog and baselines on progressively larger subsets of ADFA-U (e.g., containing 50, 500, 1000, 1500, and 2000 training samples). Due to computational constraints, we cannot perform data efficiency analysis on all datasets. Hence, we prioritize our most challenging dataset ADFA-U for this analysis. RQ3 investigates the time efficiency of FlexLog during training and inference. While employing LLM-based approaches (e.g., FlexLog) may enhance effectiveness, it often comes at the cost of increased training and inference time. This question aims to evaluate the trade-offs between effectiveness and time efficiency, providing practical insights for those considering the use of LLMs in similar tasks. Additionally, RQ3 examines the memory efficiency of FlexLog 's cache mechanism during inference, demonstrating its scalability in resource-constrained environments. Lastly, RQ4 involves exploring the impact of different configurations of FlexLog. Specifically, we assess how the exclusion of the cache $\boldsymbol{C}$, the exclusion of RAG in context-enriched prompting, and the choices of base models (e.g., removing some base models from the ensemble or replacing Mistral with other LLMs), affect the overall effectiveness or efficiency of FlexLog.

3.4.2 Experiment Setup

3.4.2.1 Datasets

We configured four datasets for ULAD from three public datasets, namely ADFA [125], LOGEVOL [13], and HDFS [97]. We exclude other popular benchmarks (e.g., BGL [98], Thunderbird [98], and Spirit [98]) because they contain only stable logs, and manually injecting instability is not feasible due to the lack of information about their annotation strategies.

Table 3.1: Overview of Datasets

Name	Sys	#Log Messages	#Anomalous Messages	#Sessions	#Log Templates	Session Length		
						avg	min	max
ADFA	Linux	2,747,550	317,388 (11.5%)	5,951	175	461.69	75	4,474
LOGEVOL	Hadoop 2	2,120,739	35,072 (1.6%)	333,699	319	6.35	1	1,963
	Hadoop 3	2,050,488	30,309 (1.4%)	343,013	313	5.97	1	1,818
	Spark 2	931,960	1,702 (0.1%)	13,892	130	67.08	1	1125
	Spark 3	1,600,273	2,430 (0.1%)	21,232	134	75.37	1	1977
HDFS	Hadoop	11,110,850	284,818 (2.9%)	575,061	48	19.32	2	30

Table 3.1 presents relevant statistics for these three datasets; column "Sys" indicates the system from which the logs were collected. "#Log Messages", "#Anomalous Messages", "#Sessions", and "#Log Templates" indicate the number of log messages, anomalous log messages, sessions, and unique log templates in each dataset, respectively. Column "Session Length" indicates the average, minimum, and maximum number of log messages in each session. We elaborate on each dataset next.

ADFA Creech et al. [125] created the Australian Defense Force Academy Linux Dataset by collecting Linux server operation logs and applying contemporary web attacks. ADFA comprises 2 747 550 log messages, i.e., Linux system calls in this context, of which 317 388 are anomalous (11.5 %). The attacks applied to the system include the exploitation of a TIKI WIKI vulnerability using a Java-based Meterpreter ("java"), password brute-forcing with the Hydra tool ("hydra"), deploying a Linux Meterpreter payload via a poisoned executable ("meter"), leveraging a remote file inclusion vulnerability to deploy a C100 webshell ("web") and creating privilege escalation by adding a superuser account with a poisoned executable ("adduser").

LOGEVOL Huo et al. [13] introduced the LOGEVOL dataset captured from the real-world operations of Hadoop 2, Hadoop 3, Spark 2 and Spark 3 systems[1]. All datasets were generated using HiBench [126] during the operation of 22 cloud computing tasks, such as sorting and classification [79, 94]. To capture real-world anomaly scenarios into logs, they injected 18 fault types into the system, including network fault, process suspension, process killing, and resource occupation. The Hadoop 2 dataset consists of 2 120 739 log messages (including 1.6 % anomalous) while the Hadoop 3 dataset is made up by 2 050 488 log messages (including 1.4 % anomalous). Notably, 104 out of 303 (33.22 %) log templates from the Hadoop 3 dataset are novel and absent from the Hadoop 2 dataset, reflecting its instability and log template evolution. The Spark 2 dataset involves 931 960 log messages, and the Spark 3 dataset is made up of 1 600 273 log messages. Compared to the Hadoop 2 and Hadoop 3 datasets, the proportion of anomalous logs in the Spark 2 and Spark 3 datasets is significantly lower, with both datasets having an anomaly rate of only 0.1 %.

[1]Hadoop versions 2.10.2 and 3.3.3 are referred to as Hadoop 2 and 3, respectively, and Spark versions 2.4.0 and 3.0.3 are denoted as Spark 2 and Spark 3, respectively.

HDFS Hadoop Distributed File System (HDFS) logs [97] were produced by running MapReduce jobs on Amazon EC2 nodes, consisting of 11 197 954 log messages, of which 284 818 (2.9 %) are anomalous. The average number of log messages in a sequence is 19.32. The total number of unique log templates is 48. This dataset includes 11 types of anomalies, such as the deletion of a block that no longer exists or receiving a block that does not belong to any file. For a comprehensive description of the anomalies, we refer readers to the original paper [97].

3.4.2.2 ULAD and SLAD Configuration

Our experiments involve the evaluation of FlexLog on both the ULAD and SLAD, with ULAD being the primary focus of this chapter and SLAD serving as a baseline in RQ1 and RQ2. As mentioned in Section 3.2.1, ULAD is characterized by the disparity between the training and testing datasets, whereas SLAD involves training and testing data drawn from the same distribution. Each dataset described in § 3.4.2.1 is configured to be used for both SLAD and ULAD, consisting of a training dataset $\mathcal{D}^{train}$ and a testing dataset $\mathcal{D}^{test}$. Additionally, a small, curated subset $\tilde{\mathcal{D}}^{train}$ is sampled from each full training dataset for the training of FlexLog. We provide details of the SLAD and ULAD configurations on each dataset next, followed by configurations of $\tilde{\mathcal{D}}^{train}$ for FlexLog.

ULAD Configuration. As discussed in Section 3.2.1, unstable logs result from system or environmental evolution. To simulate these scenarios, we configured the unstable LOGEVOL dataset LOGEVOL-U for system evolution and the unstable ADFA dataset ADFA-U for environmental evolution. To further investigate the influence of different levels of instability, we include two synthesized datasets in our experiments, namely SynHDFS-U and SYNEVOL-U. These datasets are created by injecting different levels of instability into the HDFS and LOGEVOL datasets, respectively. Table 3.2 summarizes the ULAD configuration for each dataset and presents their statistics, including the full training dataset size ($\mathcal{D}^{train}_{\#}$), FlexLog's training dataset size ($\tilde{\mathcal{D}}^{train}_{\#}$), and the testing dataset size ($\mathcal{D}^{test}_{\#}$). We also report the duplication ratio, which quantifies data leakage, i.e., log sequences appearing in both training and testing datasets. As discussed in § 3.1, data leakage allows anomaly detectors to memorize the training data, resulting in artificially inflated effectiveness on the testing data. Hence, we addressed the data leakage issues identified by Yu et al. [12].

ADFA-U We derived six ULAD configurations by splitting ADFA based on attack types. Specifically, for each configuration, five out of six attack types are used for training (e.g., $\text{ADFA}_{w/o\ java}$, in column "train" in Table 3.2, represents training data containing all attack types except the Java-based Meterpreter attack) and the remaining one for testing (e.g., $\text{ADFA}_{w/\ java}$, in the "test" column of Table 3.2, represents a testing dataset with only the Java-based Meterpreter attack), simulating external changes in real-world scenarios where novel attack types emerge during operation. Consequently, we obtain six ULAD configurations for ADFA-U involving $\text{ADFA}_{w/o\ java} \rightarrow \text{ADFA}_{w/\ java}$, $\text{ADFA}_{w/o\ hydraSSH} \rightarrow \text{ADFA}_{w/\ hydraSSH}$, $\text{ADFA}_{w/o\ hydraFTP} \rightarrow \text{ADFA}_{w/\ hydraFTP}$,

Table 3.2: ULAD Configurations

Dataset	Configuration		Duplication Ratio	#Log Sequences		
	train	test		$\mathcal{D}_{\#}^{train}$	$\tilde{\mathcal{D}}_{\#}^{train}$	$\mathcal{D}_{\#}^{test}$
ADFA-U	ADFA $_{w/o\ java}$	ADFA$_{w/\ java}$	0.32	4786	1000	1165
	ADFA$_{w/o\ hydraSSH}$	ADFA$_{w/\ hydraSSH}$	0.31	4734	1000	1217
	ADFA$_{w/o\ hydraFTP}$	ADFA$_{w/\ hydraFTP}$	0.31	4748	1000	1203
	ADFA$_{w/o\ meter}$	ADFA$_{w/\ meter}$	0.34	4835	1000	1116
	ADFA$_{w/o\ web}$	ADFA$_{w/\ web}$	0.33	4792	1000	1159
	ADFA$_{w/o\ adduser}$	ADFA$_{w/\ adduser}$	0.33	4819	1000	1132
LOGEVOL-U	Hadoop 2	Hadoop 3	0.84	302312	8558	34495
	Spark 2	Spark 3	0.50	11114	1134	4246
SYNEVOL-U	Spark 2	Spark $2_{5\%\_sequence}$	0.60	11114	1134	2778
		Spark $2_{10\%\_sequence}$	0.55			
		Spark $2_{15\%\_sequence}$	0.50			
		Spark $2_{20\%\_sequence}$	0.44			
		Spark $2_{25\%\_sequence}$	0.37			
		Spark $2_{30\%\_sequence}$	0.32			
		Spark $2_{5\%\_template}$	0.54	11114	1134	2778
		Spark $2_{10\%\_template}$	0.45			
		Spark $2_{15\%\_template}$	0.36			
		Spark $2_{20\%\_template}$	0.28			
		Spark $2_{25\%\_template}$	0.22			
		Spark $2_{30\%\_template}$	0.18			
SynHDFS-U	HDFS	SynHDFS$_{5\%_sequence}$	0.93	460048	5772	51000
		SynHDFS$_{10\%_sequence}$	0.88			
		SynHDFS$_{20\%_sequence}$	0.78			
		SynHDFS$_{30\%_sequence}$	0.69			

ADFA$_{w/o\ meter}$ →ADFA$_{w/\ meter}$, ADFA$_{w/o\ web}$ →ADFA$_{w/\ web}$,
ADFA$_{w/o\ adduser}$ →ADFA$_{w/\ adduser}$, denoted as "java", "hydraSSH", "hydraFTP", "meter", "web", and "adduser" hereafter for brevity, respectively. As reported in Table 3.2, the duplication ratio ranges from 0.31 to 0.34 in different training and testing dataset pairs, indicating that, without deduplication, approximately 31 % to 34 % log sequences in the testing datasets are already included in the training datasets.

LOGEVOL-U The LOGEVOL dataset naturally captures software evolution, namely the transition from Hadoop 2 to Hadoop 3, as well as from Spark 2 to Spark 3. These transitions result in internal changes at both template and sequence levels (defined in § 2.2.2). Hence, we use the Hadoop 2 and Spark 2 datasets for training and, correspondingly, the Hadoop 3 and Spark 3 datasets for testing. The duplication ratios for LOGEVOL Hadoop and LOGEVOL-Spark, as shown in Table 3.2, are 0.84 and 0.5, respectively; these values indicate that, without deduplication, 84% of Hadoop and 50% of Spark log sequences in the testing dataset are already included in the training dataset.

SYNEVOL-U The ULAD configurations in this dataset aim to simulate different levels of instability by applying internal changes of varying percentages to the LOGEVOL dataset. Huo et al. [13] injected log template/sequence-level changes into the LOGEVOL

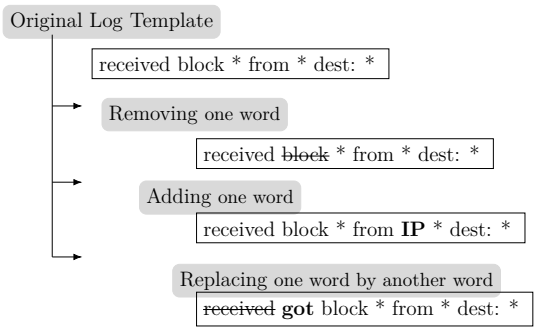


Figure 3.4: Examples of Template-level Changes

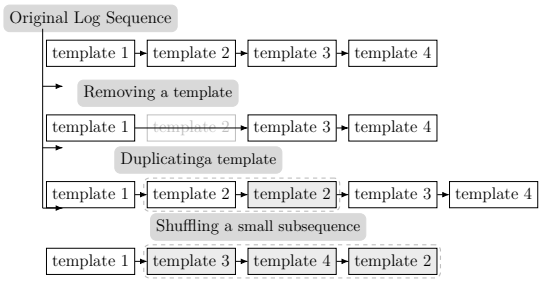


Figure 3.5: Examples of Sequence-level Changes

Spark 2 dataset, with varying injection ratio of 5 %, 10 %, 15 %, 20 %, 25 %, and 30 %. Figure 3.5 demonstrates the three types of log sequence-level changes injected, firstly introduced by Zhang et al. [14], involving removing or duplicating a log template and shuffling a small subsequence. As shown in Figure 3.4, we introduce three types of log template-level changes, including adding, removing, or replacing a word in a log template. Huo et al. [13] injected changes in the sequences in a way that sequence labels do not flip. The duplication ratio decreases from 0.6 to 0.18 as the testing set becomes more unstable.

SynHDFS-U Similar to SYNEVOL-U, we created four ULAD configurations for the HDFS dataset, namely $\text{SynHDFS}_{5\%}$, $\text{SynHDFS}_{10\%}$, $\text{SynHDFS}_{20\%}$, and $\text{SynHDFS}_{30\%}$ by changing 5%, 10%, 20%, and 30% of log sequences in the HDFS dataset, respectively. The injection ratios are determined by following common practices in the literature [14]. Similar to SYNEVOL-U, we are aware that such changes in log sequences can induce changes in their labels. We only apply sequence-level changes, excluding template-level changes due to the lack of implementation details reported by previous studies [14,38,59]. At the sequence-level, to obviate the need for re-labeling, we applied changes only to log templates that are less likely to flip the labels of the entire log sequence. These log templates are identified by a strategy proposed by Xu et al. [95], which combines building a decision tree and manual examination. To reduce the cost of manual examination, we sampled and applied changes to a subset of the HDFS dataset instead of the full dataset. Concretely, we randomly selected 50,000 normal and 1,000 anomalous log sequences, following the study by Zhang et al. [14], to keep the anomaly percentage (2 %) close to that of the original HDFS dataset. The duplication ratio decreases from 0.93 to 0.69 as the testing set becomes more unstable.

SLAD Configuration. For the ADFA dataset, we drew the training and testing data from the full dataset, containing all six types of anomalies. For LOGEVOL Hadoop 2, Spark 2, and HDFS, we adopt the same training datasets as in their ULAD configurations, whereas the testing datasets differ. While ULAD employs unstable testing data, SLAD uses stable testing data collected from the same system as the training data, specifically from Hadoop 2, Spark 2, and HDFS operations, respectively. Notably, the testing dataset for HDFS SLAD configuration is the same as the one used in its ULAD configuration, a

Table 3.3: Overview of Baselines

Learning Method	Approach	Parser	Log Representation	ML Method	Base Model
Unsupervised	PCA	Yes	Template ID	Traditional ML	PCA
	LogCluster	Yes	Template ID	Traditional ML	Clustering
	DeepLog	Yes	Template ID	Deep Learning	LSTM
	LogAnomaly	Yes	Template2Vec	Deep Learning	LSTM
Semi-supervised	PLELog	Yes	FastText and TF-IDF	Deep Learning	GRU
Supervised	LogRobust	Yes	FastText and TF-IDF	Deep Learning	BiLSTM
	CNN	Yes	Logkey2vec	Deep Learning	CNN
	NeuralLog	No	BERT	Deep Learning	Transformer
	LightAD	Yes	TemplateID	Traditional ML & Deep Learning	KNN, DT, SLFN

subset sampled from the original testing dataset, but without instability injection. This ensures consistency and a fair comparison between the ULAD and SLAD configuration of HDFS. Also, we did not configure LOGEVOL Hadoop 3 and Spark 3 for SLAD as ULAD because the evolution information from Hadoop 3 and Spark 3 to other versions was not available.

Training Dataset Configuration for FlexLog. In RQ1, we compare FlexLog and the baselines with their respective optimal settings. Baselines are trained on the full training datasets $\mathcal{D}^{train}$, following implementations in their original papers, whereas FlexLog is trained on small subsets $\tilde{\mathcal{D}}^{train}$ randomly sampled from $\mathcal{D}^{train}$. As reported in the second-to-last column ("$\tilde{\mathcal{D}}^{train}_{\#}$") of Table 3.2, their data sizes are determined empirically for each dataset to achieve the optimal performance of FlexLog. For small datasets with low duplication ratios such as ADFA-U, we randomly selected 1 000 log sequences from their full training dataset. For larger datasets with high duplication ratios, such as LOGEVOL-U Hadoop, LOGEVOL-U Spark, SYNEVOL-U, and SynHDFS-U, unique anomalous log sequences are rare, accounting for only 0.2 % to 2 % of the full datasets, respectively. To maximize the use of these rare anomalous log sequences, we included all of them in the fine-tuning datasets and sampled 20 % of the unique normal log sequences, thereby preventing excessive duplication.

3.4.2.3 Baselines

We considered nine ML methods as baselines in this chapter, including four unsupervised, one semi-supervised, and four supervised. Among these methods, LightAD [96] achieves the best performance on the SLAD task. However, the leading approach for ULAD remains undetermined as different evaluation datasets are used in reported studies. Our choice of baselines is also determined by source code availability to ensure the reliability of the implementation. Consequently, we had to exclude models such as SwissLog [59], HitAnomaly [38], EvLog [13], and LLMeLog [99]. Our implementations are based on the code provided by Yu et al. [96], Le et al. [110], and He et al. [43]. We have also not included LogPrompt [100] in our evaluation since it relies on anomaly detection at the message level, ignoring sequential characteristics such as temporal dependencies, whereas our datasets are labeled at the sequence level. Although FlexLog is supervised, we in-

clude semi-supervised and unsupervised baselines in the comparison as they generally incur lower training and labeling costs, while still providing a meaningful reference for assessing the overall effectiveness of our model.

Table 3.3 shows the main characteristics of the baselines; we provide a brief description in the following. *Principal Component Analysis (PCA)* [95], a dimensionality reduction method, converts logs into count vectors [42] and then uses the PCA algorithm to detect the label of log sequences by assigning them to either the normal or anomalous space. In this chapter, by PCA we refer to the PCA-based model introduced by Xu et al. [95] as an anomaly detector. *LogCluster* [94] clusters log sequences by computing the similarity of log representations to the centroid of normal logs. *DeepLog* [11] applies two layers of Long Short-Term Memories (LSTMs) in their network [24] to predict the next event from a given log sequence and labels sequences as anomalous if the predicted log is different than the actual log template. *LogAnomaly* [16] has an architecture similar to DeepLog, but is further improved by adopting semantic embeddings for log templates and adding an attention layer between LSTM layers. *PLELog* [51] is a semi-supervised strategy that uses normal data as well as a small subset of unlabeled data to train. First, it adopts a clustering method (HDBSCAN [127]) to probabilistically predict the labels of unlabeled data and then uses them to train an attention-based GRU [128] to detect anomalies. *LogRobust* [14] uses a pre-trained word vectorizer (FastText [129]) to extract semantic information from log templates and utilizes an attention-based BiLSTM model [25] to detect anomalous log sequences. *CNN* [22] transforms an input log sequence into a trainable matrix and uses this matrix as input to train a Convolutional Neural Network [35, 130] for log-based anomaly detection. *NeuralLog* [37] extracts the semantic meaning of raw log messages and represents them as semantic vectors, which are then used to detect anomalies through a transformer-based classification model [27]. *LightAD* [96] employs Bayesian method to select the most effective model from a heterogeneous pool of ML/DL algorithms—including KNN [131], DT [132], and SLFN [133]—while simultaneously optimizing hyperparameters for the SLAD task. To ensure fair comparisons, we adopted the same model pool and employed a small, held-out validation dataset to identify the optimal model for each dataset as instructed in Yu et al. [96]. The performance of the optimal model, evaluated on test data, serves as LightAD's reported effectiveness.

3.4.2.4 Evaluation Metrics and Statistical Testing

To provide a comprehensive evaluation, we assess FlexLog in terms of effectiveness, data efficiency, and time efficiency. Furthermore, we investigate the statistical significance of differences (from the perspectives of effectiveness and time efficiency) on each dataset.

Effectiveness To measure the effectiveness of FlexLog, we use precision, recall, and F1 score as metrics. We consider TP (true positive) as the number of anomalies that are correctly detected by the model, FP (false positive) as the number of normal log sequences that are labeled as anomalous by the model, and FN (false negative) as the number of anomalous log sequences that the model fails to identify. Precision (P) is calculated by $\frac{TP}{TP+FP}$ as the percentage of true anomalies among all anomalies detected by the model.

Recall (R) is the proportion of actual anomalies detected, computed by $\frac{TP}{TP+FN}$. F1 score (F1) is the harmonic mean of precision and recall, i.e., $2 * \frac{P*R}{P+R}$.

Data Efficiency We define data efficiency to be the ability of a method to achieve accurate results while minimizing the use of labeled data for training. Considering a training dataset with $\mathcal{D}_{\#}$ log sequences, we quantify the usage of labeled data using $\mathcal{U}_{\#}$, which represents the number of unique log sequences. Each unique log sequence corresponds to a distinct pattern that requires annotation, meaning that a higher $\mathcal{U}_{\#}$ reflects greater labeling effort.

A data-efficient method, such as FlexLog, requires only a small subset of the full dataset for training, reducing the overall usage of labeled data. To compare data efficiency across different methods, we introduce the relative metric $\mathcal{U}_{\%}$ as in Equation 3.7, which measures the percentage of unique log sequences in the subset relative to the total unique log sequences in the full dataset (denoted by $\mathcal{U}_{\#}^{full}$).

$$\mathcal{U}_{\%} = \frac{\mathcal{U}_{\#}}{\mathcal{U}_{\#}^{full}} \tag{3.7}$$

To quantify the reduction in labeled data achieved by data-efficient methods compared to methods trained on full datasets, we define the labeled data usage reduction $\Delta\mathcal{U}_{\%}$ as in Equation 3.8. A higher $\Delta\mathcal{U}_{\%}$ indicates a greater reduction, demonstrating the superior data efficiency of the method under evaluation, and vice versa.

$$\Delta\mathcal{U}_{\%} = 1 - \mathcal{U}_{\%} \tag{3.8}$$

Time Efficiency We evaluate the time efficiency in terms of training and inference time for each model. For training time, we calculate the total training time taken for a model. For inference time, we calculate the average inference time for one input sequence in the testing set.

Memory Efficiency We evaluate memory efficiency based on the additional memory required by the cache component during inference. Specifically, we measure the memory overhead introduced by the in-memory cache, which stores predictions for all previously seen unique sequences. To approximate this overhead, we compute the memory consumed by the cache in the structure of a dictionary after processing the entire test set. Since the cache grows incrementally with the number of unique sequences, this measurement represents its maximum size at the end of inference.

Statistical Testing To mitigate the potential influence of randomness on our results, we repeat each experiment on each configuration 5 times and report the average performance across the runs. This ensures our analysis is robust and not unduly influenced by any single random sampling or stochastic training and fine-tuning, providing a reliable evaluation. We further perform Mann-Whitney U tests, as recommended in Arcuri et al. [134], to compare

different models and configurations on each dataset, resulting in test group sizes of 30 for ADFA-U (6 configurations), 10 for LOGEVOL-U (2 configurations), 20 for SynHDFS-U (4 configurations), and 30 for SYNEVOL-U (6 configurations).

The Mann-Whitney U test is a non-parametric statistical test that compares two methods, A and B, without any assumption of the data distribution. It computes a p-value, which indicates whether the observed performance difference is statistically significant. The null hypothesis presumes no significant distinction between performance A and B. If the p-value falls below the commonly used threshold of 0.05, we reject the null hypothesis and conclude that the difference is statistically significant. Conversely, if the p-value is greater than or equal to 0.05, the difference is considered non-significant, meaning the observed difference could be due to randomness.

3.4.2.5 Other Settings.

We conducted all experiments on a cloud computing environment containing 28 CPU cores for computation, 2 × Nvidia L40S GPU devices, and 256 GB RAM.

3.4.3 Implementation

In this section, we introduce the implementation details of preprocessing, FlexLog, and the baselines.

3.4.3.1 Preprocessing

Two primary steps of preprocessing are parsing and partitioning, as described in Section 3.3.

Log parsing is used to provide structured context for FlexLog (as explained in § 3.3.1) and log-parsing-based baselines such as LogRobust and CNN. For ADFA-U, parsing is not required since each log message is a one-word system call. For datasets with evolving templates—ADFA-U and SYNEVOL-U —we follow their original authors' practice and use the Prefix Graph parser [135]. This parser does not require a training set and is more flexible in handling varying template lengths and substructures compared to fixed-depth approaches such as Drain [81]. For SynHDFS-U, we use Drain following the common practice for this specific synthetic dataset [14, 38][2]. SynHDFS-U exhibits instability at the sequence level, but its log templates remain stable; hence, the limitations of Drain in handling evolving templates do not apply in this case. Since each log message in ADFA consists of a one-word system call, the subsequent RAG can easily associate a log message with its relevant information, such as the description of that system call.

[2]We acknowledge that on the original HDFS dataset, more recent log parsers [103, 136] demonstrated higher parsing effectiveness than Drain. However, as a recent study [137] demonstrated, there is no correlation between parsing accuracy and anomaly detection accuracy.

Applying a smaller window over long sequences facilitates the localization of anomalies when logs are labeled at the message level. Hence, we applied sliding window-based partitioning with a window size of 50 on LOGEVOL-Hadoop. In contrast, sliding window partitioning is not an option for long sequences in the ADFA, LOGEVOL-Spark, SYNEVOL-U, and HDFS datasets due to the absence of message-level labels. For HDFS, most sessions are short, with only 3.5 % sessions exceeding 30 templates. We followed the implementation of Le et al. [110] and truncated these long sessions to ensure that all sessions were within the 30-template limit.

3.4.3.2 FlexLog

The implementation of FlexLog mainly involves three key aspects, namely the selection of base models for ensemble learning, the fine-tuning of the LLM base model, and the hyperparameter settings of ML base models in FlexLog.

Base Model Selection FlexLog combines multiple heterogeneous base models through ensemble learning, including ML/DL models and LLMs. Specifically, the ensemble in FlexLog comprises three ML base models (KNN, DT, and SLFN) with one LLM base model (Mistral [138]). We selected KNN, DT, and SLFN due to their high effectiveness on SLAD task as reported by Yu et al. [96]. We selected the LLM base model (Mistral 22B) through the empirical evaluation of multiple open-source and closed-source LLMs. To elaborate, closed-source LLMs, such as GPT-4o, are considered state-of-the-art LLMs in various domains, albeit at a high cost [139]. We experimented with a major version—GPT-4o (`GPT-4o-turbo` version) — based on OpenAI's recommendation in terms of performance [113, 140]. In contrast, open-source LLMs incur no cost and offer more flexibility regarding fine-tuning and inference. Within the limit of our computing resources, we explored two open-source LLMs that have shown competitive performance to closed-source LLMs [141, 142]: LLama 3.1 8B and Mistral 22B.

Fine-tuning LLMs For open-source LLMs, we utilized 4-bits QLora [120] for fine-tuning, with the following configurations based on our preliminary experiments: rank=16, alpha=16, and batch=1. We fine-tuned Llama 3.1 8B and Mistral Small 22B using the Unsloth library for its efficiency [143][3]. The number of steps is empirically tuned for unique pairs of LLM and dataset separately, using grid-search and cross-validation; values range from 500 to 2500 in steps of 500. For the closed-source LLM of GPT-4o, fine-tuning was accomplished through the OpenAI API, which incurred an associated cost. Hyperparameters such as the number of epochs and batch size for fine-tuning GPT were optimized and automatically determined by OpenAI fine-tuning APIs on each dataset.

LLMs, even after fine-tuning, tend to generate non-deterministic output, which threatens the reliability of FlexLog for ULAD. To ensure reliable anomaly detection, we instruct the LLMs to generate the response with minimum temperature (e.g., 0.1 for Mistral). In case the responses deviate from explicit labels (e.g., "normal", "anomalous", "0" or "1"), ambiguous responses trigger up to five regeneration attempts with progressively higher

[3]The corresponding Hugging Face model names are `unsloth/Meta-Llama-3.1-8B-bnb-4bit` and `unsloth/Mistral-Small-Instruct-2409-bnb-4bit`.

temperatures from 0.2 to 1, in steps of 0.2, to diversify outputs. If no valid label is parsed after all attempts, we classify the sequence as "normal" to reduce false positives, which might trigger alert fatigue and operational disruption unnecessarily.

Hyperparameter settings of ML base models in FlexLog For the DT and SLFN base models, we use the default values provided by Scikit-learn Library across all datasets. Our preliminary experiments suggest that tuning these hyperparameters with limited data often leads to overfitting and reduced effectiveness compared to the default settings. While further tuning could potentially improve FlexLog's performance, we leave this for future work. Specifically for DT, we set criterion to "gini", max_depth to "None", and min_samples_split to 2. For SLFN, we set hidden_layer_sizes to 100, activation to "relu", solver="adam", and batch_size to "auto". For KNN, since the number of neighbors plays an important role in handling imbalanced datasets [144], we empirically tuned it with grid search and cross-validation on limited training data. We set the number of neighbors to 2 for ADFA and LOGEVOL Hadoop, and 1 for HDFS[4]. For the extremely imbalanced datasets—LOGEVOL Spark and SYNEVOL-U, which share the same training set—KNN performs poorly on the validation set, exhibiting significantly lower effectiveness compared to DT and SLFN. This aligns with known limitations of KNN on highly imbalanced datasets, where the majority class tends to dominate the predictions [145]. Therefore, we excluded KNN from FlexLog for LOGEVOL Spark and SYNEVOL-U.

Lastly, for the ensemble strategy, we implemented a majority voting algorithm in which, in the event of a tie, the sequence is labeled as normal, as described in § 3.3.4. Our preliminary results across all four datasets indicate that this strategy remains the most effective and straightforward compared to alternative ensemble learning approaches, including SNAIL [146] and MetaFormer [147] (see Appendix A.1).

3.4.3.3 Baselines

For baselines, we set hyperparameters as reported in their original papers or suggested by their implementation packages. When hyperparameters were not available in either of them, particularly for datasets such as ADFA, we empirically tuned the parameters by grid search with a cross-validation approach. For LightAD, we fine-tuned hyperparameters using Bayesian optimization, available in their implementation code for KNN, DT, and SLFN. LogAnomaly's representation model (template2vec) requires domain-specific antonyms and synonyms for training. This information is not available in the original paper, and thus, similar to the method previously adopted [110], we used a pre-trained FastText model [129] to compute the semantic vectors. As LOGEVOL Hadoop and Syn-HDFS training sets are too large to process on NeuralLog, we used a subset containing the first 200 000 log messages, following the methodology adopted in prior work [110].

[4]Overall, due to the limited training data, we recommend using default parameters and tuning only those that are highly sensitive to imbalanced data, as anomalies are often rare in real-world datasets, using cross-validation. In the future, we plan to conduct more experiments to explore the correlation between the percentage of anomalous data and the optimal number of neighbors.

Table 3.4: Statistics of training data for FlexLog and baselines used in RQ1 on ADFA-U, LOGEVOL, SYNEVOL-U and SynHDFS-U.

Dataset	$\mathcal{D}_{\#}$		$\mathcal{U}_{\#}$		$\mathcal{U}_{\%}$		$\Delta\mathcal{U}_{\%}$
	Baselines	FlexLog	Baselines	FlexLog	Baselines	FlexLog	
ADFA-U	4 785	1 000	3 181	686	100 %	21.57%	78.43 pp
LOGEVOL-U	313 426	9 692	29 809	9 692	100 %	32.51%	67.49 pp
SYNEVOL-U	11 114	1 134	4 939	1 134	100 %	22.96%	77.04 pp
SynHDFS-U	460 048	5 772	15 545	5 772	100 %	37.13%	62.87 pp

3.4.4 Data availability.

The replication package, including our synthesized datasets, additional experiment results, and source code, is publicly available [148].

3.5 Results

3.5.1 RQ1: Overall Effectiveness

RQ1 investigates the effectiveness of FlexLog trained with limited data and compares it to baselines trained with full datasets on ULAD and SLAD. This comparison between FlexLog and baselines is notably less advantageous for FlexLog, as it is trained with less data than the baselines[5]. To prevent inflated effectiveness caused by data leakage, test data is deduplicated from the training data, as explained in § 3.3.4; a comparison of model performance evaluated on the test set with and without deduplication is also provided in Appendix A.2.

Table 3.4 reports the configurations of training datasets for FlexLog and baselines, including the training dataset size ($\mathcal{D}_{\#}$), the number of unique log sequences ($\mathcal{U}_{\#}$), the percentage of unique log sequences relative to the total unique log sequences in the full datasets ($\mathcal{U}_{\%}$) and the reduction in percentage points in labeled data achieved by FlexLog ($\Delta\mathcal{U}_{\%}$); please refer to § 3.4.2.4 for details of these metrics. In the rest of this section, we first report the evaluation of the overall effectiveness of FlexLog and baselines on two real-world unstable datasets, LOGEVOL-U and ADFA-U (§ 3.5.1.1). To further analyze the influence of instability, we conducted controlled experiments on two synthesized datasets, SYNEVOL-U and SynHDFS-U, where varying levels of instability are systematically introduced (§ 3.5.1.2).

3.5.1.1 Effectiveness on Real-world Datasets

Table 3.5 and Table 3.6 report the effectiveness of FlexLog on two real-world datasets, LOGEVOL and ADFA, respectively. Column *"Data"* specifies different dataset configu-

[5]The effectiveness of baselines when trained on the same limited data as FlexLog is reported in Appendix A.3.

Table 3.5: Effectiveness of FlexLog and baselines for ULAD and SLAD on the ADFA dataset

Data	Unstable	M	limited data	full training set								
				Supervised				Semi-S	Unsupervised			
			FlexLog	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
ADFA	No	P	0.708	0.820	0.538	0.718	0.666	0.736	0.357	0.334	0.255	0.196
		R	0.894	0.814	0.602	0.708	0.842	0.216	0.430	0.523	0.907	0.139
		F1	0.791	**0.817**	0.568	0.713	0.744	0.334	0.390	0.408	0.398	0.162
adduser	Yes	P	0.619	0.673	0.303	0.711	0.547	0.507	0.208	0.198	0.217	0.139
		R	0.857	0.786	0.651	0.415	0.775	0.281	0.483	0.562	0.725	0.202
		F1	0.718	**0.725**	0.412	0.524	0.641	0.361	0.291	0.292	0.334	0.165
hydraFTP	Yes	P	0.686	0.780	0.532	0.612	0.882	0.247	0.345	0.411	0.268	0.139
		R	0.913	0.731	0.306	0.306	0.653	0.897	0.650	0.581	0.950	0.133
		F1	**0.784**	0.754	0.388	0.408	0.750	0.388	0.451	0.481	0.418	0.144
hydraSSH	Yes	P	0.657	0.833	0.298	0.656	0.882	0.522	0.408	0.390	0.299	0.121
		R	0.804	0.635	0.479	0.262	0.653	0.433	0.583	0.554	0.999	0.121
		F1	**0.723**	0.666	0.368	0.374	0.750	0.473	0.480	0.458	0.461	0.140
java	Yes	P	0.634	0.699	0.366	0.695	0.752	0.371	0.357	0.269	0.213	0.169
		R	0.650	0.590	0.849	0.457	0.549	0.513	0.516	0.459	0.959	0.157
		F1	0.642	0.639	0.511	0.558	0.635	0.430	0.422	0.339	0.348	0.158
web	Yes	P	0.639	0.788	0.330	0.583	0.786	0.799	0.254	0.335	0.186	0.204
		R	0.709	0.487	0.741	0.395	0.513	0.136	0.530	0.373	0.829	0.191
		F1	**0.672**	0.602	0.457	0.449	0.621	0.233	0.343	0.353	0.304	0.197
meter	Yes	P	0.564	0.710	0.312	0.745	0.642	0.569	0.147	0.178	0.235	0.210
		R	0.859	0.732	0.889	0.577	0.799	0.163	0.425	0.436	0.943	0.139
		F1	**0.682**	0.679	0.461	0.651	0.711	0.253	0.218	0.253	0.175	0.241
average	Yes	P	0.633	0.747	0.357	0.667	0.748	0.503	0.286	0.297	0.236	0.164
		R	0.799	0.660	0.652	0.402	0.657	0.404	0.531	0.494	0.901	0.157
		F1	**0.704**	0.677	0.433*	0.494*	0.685	0.356*	0.368*	0.363*	0.340*	0.174*

* FlexLog yields a significant higher F1-score than compared baseline.

rations, including SLAD alongside with various ULAD settings. For instance, $\text{Hadoop}_{2\rightarrow3}$ represents that logs produced from Hadoop 2 and Hadoop 3 are used as training and testing datasets, respectively. Column "Unstable" indicates whether the data configuration is unstable or not. Column *"M"* reports the effectiveness metrics introduced in § 3.4.2.4. We recall in this table that FlexLog is trained with only *"limited data"* while all baselines are all trained with the *"full training dataset"*, including *"supervised"*, *"semi-supervised"* and *"unsupervised"* baselines.

As shown in Table 3.5, in the SLAD of the ADFA dataset, FlexLog achieves an F1 score of 0.791, second only to LightAD (0.817) among all methods. However, in the ULAD, where log instability arises from environmental changes, i.e., the introduction of novel attack types, FlexLog outperforms all baselines on 5 out of 6 configurations. The only exception occurs in the *adduser configuration*, where LightAD surpasses FlexLog with a small margin of 0.7 pp (72.5% − 71.8%). Overall, FlexLog yields the highest average F1 score of 0.704 in ULAD configurations, followed by CNN (0.685) and LightAD (0.677). A Mann-Whitney U test indicates that the differences between the average F1 scores of FlexLog and LightAD, as well as the differences between FlexLog and CNN, are statistically insignificant, whereas FlexLog outperforms all other baselines significantly. Moreover, when moving from SLAD to ULAD, the most effective baseline in SLAD (LightAD) experiences an average F1 score decrease of 14 pp, whereas the decrease with FlexLog remains below 9 pp, alleviating the impact of unstable logs on anomaly

detection effectiveness.

Notably, FlexLog achieves such high effectiveness on the ADFA dataset with only limited training data, while baselines like LightAD and CNN are trained with full datasets. As shown in Table 3.4, FlexLog's fine-tuning datasets contain only 21.57 % of unique log sequences relative to the total unique log sequences in the full ADFA datasets, each requiring a dedicated label. This translates to a reduction of 78.43 pp in usage of labeled data while achieving state-of-the-art effectiveness on the ADFA dataset.

We observe similar results on the LOGEVOL-U dataset, where instability is introduced due to system evolution, e.g., software system version updates. As shown in Table 3.6, we evaluated two SLAD configurations ($\text{Hadoop}_{2\to2}$ and $\text{Spark}_{2\to2}$) and two ULAD configurations ($\text{Hadoop}_{2\to3}$ and $\text{Spark}_{2\to3}$). In the $\text{Hadoop}_{2\to2}$ SLAD configuration, all supervised approaches achieve a high F1 score (≥ 0.980), including FlexLog, LightAD, NeuralLog, LogRobust, and CNN. In the other SLAD configuration $\text{Spark}_{2\to2}$, FlexLog surpass baselines, reaching an F1 score of 0.984 by 1.4 pp from the top baseline, LightAD. In the $\text{Hadoop}_{2\to3}$ and $\text{Spark}_{2\to3}$ ULAD configurations, FlexLog yields F1 scores of 0.982 and 0.892, respectively, remaining the best approach compared to all the baselines. On average, FlexLog outperforms all the baselines in terms of F1 score, with a minimum margin of 1.8 pp ($0.928 - 0.910$). Additionally, when moving from SLAD to ULAD, the most effective baseline in SLAD (LightAD) experiences an average F1 score decrease of 10 pp, whereas the decrease with FlexLog remains below 7 pp, alleviating once again the impact of unstable logs on anomaly detection effectiveness.

Similar to the ADFA-U dataset, FlexLog achieves such high effectiveness on the LOGEVOL dataset by training on a relatively limited dataset instead of the entire LOGEVOL training dataset. Specifically, as depicted in Table 3.4, FlexLog's fine-tuning dataset contain only 32.51 % of unique log sequences relative to the total unique log sequences in the full LOGEVOL datasets, indicating a reduction in usage of labeled data of 67.49 pp.

3.5.1.2 Impact of Log Instability

As detailed in § 3.4.2.2, we conducted comprehensive experiments on two synthesized datasets, SynHDFS-U and SYNEVOL-U, to analyze the impact of instability at both the log sequence and log template levels. Specifically, SYNEVOL-U exhibits instability at both the sequence and template levels, whereas SynHDFS-U is characterized solely by sequence-level instability.

Instability at Log Sequence Level. Table 3.7 presents the performance of FlexLog and baseline approaches on SynHDFS-U with sequence-level instability. As the injection ratio of changes increases, FlexLog consistently achieves the highest F1 score across all unstable configurations, demonstrating its robustness to varying levels of instability. On average, FlexLog outperforms all baselines, exceeding the top-performing baseline, LightAD, by a margin of 1.2 pp in F1 score. A Mann-Whitney test reveals that the difference in F1 scores between FlexLog and LightAD is statistically insignificant, indicating comparable performance between the two. However, unlike all baselines—including

Table 3.6: Effectiveness of FlexLog and baselines for ULAD and SLAD on the LOGEVOL dataset

Data	Unstable	M	limited data	full training set								
				Supervised				Semi-S	Unsupervised			
			FlexLog	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
$Hadoop_{2\to2}$	No	P	0.999	0.999	0.997	0.984	0.997	0.648	0.263	0.384	0.952	0.267
		R	0.986	0.994	0.986	0.976	0.997	0.888	0.616	0.352	0.320	0.867
		F1	0.993	**0.997**	0.992	0.980	**0.997**	0.749	0.368	0.367	0.479	0.408
$Spark_{2\to2}$	No	P	0.999	0.999	0.999	0.941	0.999	0.172	0.501	0.512	0.771	0.072
		R	0.969	0.939	0.636	0.969	0.878	0.129	0.393	0.443	0.818	0.471
		F1	**0.984**	0.968	0.777	0.952	0.935	0.243	0.441	0.475	0.794	0.125
$Hadoop_{2\to3}$	Yes	P	0.998	0.998	0.914	0.905	0.992	0.626	0.221	0.384	0.510	0.225
		R	0.965	0.963	0.984	0.950	0.968	0.819	0.522	0.352	0.371	0.898
		F1	**0.982**	0.980	0.948	0.927	0.980	0.709	0.310	0.367	0.430	0.360
$Spark_{2\to3}$	Yes	P	0.999	0.981	0.916	0.696	0.992	0.105	0.141	0.08	0.347	0.061
		R	0.805	0.708	0.766	0.832	0.736	0.377	0.458	0.606	0.805	0.484
		F1	**0.892**	0.829	0.834	0.757	0.840	0.165	0.216	0.122	0.485	0.108
Average ($Spark_{2\to3}$, $Hadoop_{2\to3}$)	Yes	P	0.998	0.99	0.915	0.786	0.992	0.366	0.181	0.232	0.428	0.143
		R	0.871	0.83	0.875	0.863	0.852	0.887	0.49	0.479	0.588	0.691
		F1	**0.928**	0.898*	0.891*	0.833*	0.910*	0.437*	0.263*	0.244*	0.458*	0.234*

* FlexLog yields a significant higher F1-score than baseline.

LightAD—which are trained on the full dataset, FlexLog is trained on only a small subset comprising 37.13 % of unique log sequences (Table 3.4). This means that FlexLog achieves similar effectiveness and robustness while reducing the usage of labeled data by 62.87 pp. The effectiveness of FlexLog and baselines on SYNEVOL-U under varying log sequence instability demonstrated similar results to SynHDFS-U that we have provided in Appendix A.4.

Instability at Log Template Level. Table 3.8 reports the effectiveness of FlexLog compared with baselines on SYNEVOL-U under varying template-level changes. Once again, FlexLog achieves the highest precision, recall, and F1 score across all injection ratios, outperforming the top-performing baseline—LightAD—by 1.9 pp (96.3% – 94.4%) in terms of average F1 score while reducing usage of labeled data by 77.04 pp. A Mann-Whitney U test suggests this difference is statistically significant. Similar to sequence-level changes in SYNEVOL-U, increasing template-level instability negatively impacts the effectiveness of semi-supervised and unsupervised approaches, whereas supervised methods—FlexLog, LightAD, NeuralLog, LogRobust, and CNN—are relatively robust in terms of effectiveness across instability levels. This is likely because many template modifications in SYNEVOL-U involve minor textual variations, such as inserting, removing, or replacing short tokens, which have minimal impact on ULAD, especially in long log sequences with up to 1 818 templates. For instance, a log template stating "`SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(root); groups with view permissions: Set(); users with modify permissions: Set(root); ...`" was modified by inserting `"so"` before `"users with modify permissions"`, a minor change that has limited impact on ULAD effectiveness.

Table 3.7: Effectiveness of FlexLog and baselines under different sequence-level injection ratios on SynHDFS-U.

Data	Unstable	M	limited data	full training set								
				Supervised				Semi-S	Unsupervised			
			FlexLog	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
0%	No	P	0.954	0.976 3	0.949	0.957	0.933	0.630	0.243	0.725	0.999	0.924
		R	0.999	0.990	0.985	0.980	0.951	0.966	0.971	0.927	0.346	0.667
		F1	0.976	**0.983**	0.966	0.969	0.942	0.763	0.389	0.814	0.514	0.762
5%	Yes	P	0.953	0.957	0.944	0.995	0.932	0.602	0.236	0.678	0.986	0.976
		R	0.995	0.985	0.985	0.979	0.952	0.554	0.961	0.894	0.346	0.667
		F1	**0.974**	0.971	0.964	0.878	0.944	0.577	0.380	0.771	0.512	0.792
10%	Yes	P	0.954	0.966	0.914	0.692	0.933	0.404	0.239	0.649	0.999	0.225
		R	0.995	0.980	0.980	0.974	0.951	0.884	0.975	0.927	0.350	0.673
		F1	**0.974**	0.973	0.946	0.809	0.942	0.555	0.385	0.764	0.519	0.337
20%	Yes	P	0.949	0.943	0.906	0.560	0.933	0.345	0.482	0.644	0.949	0.158
		R	0.995	0.966	0.980	0.956	0.951	0.798	0.538	0.889	0.360	0.694
		F1	**0.971**	0.954	0.942	0.707	0.942	0.482	0.509	0.747	0.522	0.258
30%	Yes	P	0.940	0.925	0.896	0.489	0.929	0.243	0.464	0.548	0.915	0.137
		R	0.985	0.956	0.970	0.951	0.947	0.877	0.572	0.903	0.365	0.718
		F1	**0.964**	0.940	0.931	0.646	0.938	0.381	0.512	0.682	0.522	0.230
Average	Yes	P	0.949	0.948	0.915	0.684	0.932	0.398	0.355	0.63	0.962	0.374
		R	0.992	0.972	0.979	0.965	0.95	0.778	0.762	0.903	0.355	0.688
		F1	**0.971**	0.959	0.946*	0.760*	0.942*	0.499*	0.446*	0.741*	0.519*	0.404*

* FlexLog yields a significant higher F1-score than compared baseline.

The answer to RQ1 is that FlexLog achieves **state-of-the-art effectiveness** on both real-world and synthesized datasets while exhibiting **high data efficiency** (with a reduction in labeled data usage ranging between 62.87 pp and 78.43 pp). On synthesized datasets, FlexLog remains effective under up to 30 % sequence- and template-level instability.

3.5.2 RQ2: Data Efficiency on the ADFA-U Dataset

RQ2 focuses on the data efficiency analysis of FlexLog and baselines. As mentioned in § 3.4.1, computational constraints preclude us from investigating the data efficiency on all the datasets. Hence, we focus on ADFA-U in this RQ since it includes six diverse configurations, enabling a robust evaluation under different real-world ULAD scenarios. As shown in RQ1, these diverse configurations make it the most challenging dataset in our experiments in terms of detection effectiveness. Specifically, for each configuration, we randomly sample five training subsets (50, 500, 1000, 1500, and 2000 samples) to simulate different levels of data scarcity.

Table 3.9 reports statistics for the sampled subsets, including the percentage of unique log sequences ($\mathcal{U}_{\%}$), which is calculated by dividing the number of unique log sequences in each subset by the total number of unique log sequences in the full training dataset. This percentage represents the proportion of labeling effort required for each subset compared to full dataset annotation. The table further provides the percentages of unique log sequences for each class ($\mathcal{U}^{+}_{\%}$ and $\mathcal{U}^{-}_{\%}$). Notably, the smallest subset ($\mathcal{D}_{\#} = 50$) contains $< 1.5\%$ of unique log sequences, representing extreme data scarcity scenarios where the availability of labeled data is highly limited.

Table 3.8: Effectiveness of FlexLog and baselines under different template-level injection ratios on SYNEVOL-U.

Data	Unstable	M	limited data	full training set								
				supervised				Semi-S	Unsupervised			
			FlexLog	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
0%	No	P	0.999	0.999	0.999	0.941	0.999	0.172	0.501	0.512	0.771	0.072
		R	0.969	0.939	0.636	0.969	0.878	0.129	0.393	0.443	0.818	0.471
		F1	**0.984**	0.968	0.777	0.952	0.935	0.243	0.441	0.475	0.794	0.125
5%	Yes	P	0.999	0.999	0.999	0.943	0.999	0.127	0.351	0.282	0.651	0.062
		R	0.970	0.941	0.647	0.971	0.882	0.315	0.393	0.382	0.823	0.500
		F1	**0.985**	0.969	0.785	0.956	0.937	0.181	0.440	0.325	0.727	0.111
10%	Yes	P	0.999	0.999	0.999	0.921	0.937	0.120	0.260	0.181	0.622	0.054
		R	0.942	0.914	0.600	0.969	0.857	0.316	0.371	0.400	0.800	0.457
		F1	**0.970**	0.955	0.750	0.944	0.895	0.174	0.305	0.250	0.700	0.097
15%	Yes	P	0.999	0.999	0.999	0.944	0.916	0.117	0.265	0.180	0.604	0.054
		R	0.921	0.894	0.605	0.918	0.891	0.315	0.447	0.447	0.763	0.447
		F1	**0.958**	0.944	0.754	0.931	0.904	0.171	0.333	0.257	0.674	0.097
20%	Yes	P	0.999	0.999	0.916	0.943	0.906	0.112	0.180	0.095	0.457	0.047
		R	0.941	0.911	0.647	0.951	0.852	0.315	0.382	0.411	0.794	0.470
		F1	**0.969**	0.953	0.758	0.946	0.878	0.165	0.245	0.155	0.580	0.085
25%	Yes	P	0.999	0.999	0.999	0.908	0.916	0.127	0.188	0.100	0.507	0.051
		R	0.904	0.850	0.625	0.954	0.846	0.239	0.400	0.400	0.800	0.475
		F1	**0.950**	0.918	0.769	0.930	0.880	0.166	0.256	0.160	0.621	0.093
30%	Yes	P	0.973	0.999	0.931	0.923	0.947	0.121	0.180	0.096	0.438	0.053
		R	0.925	0.857	0.642	0.954	0.857	0.242	0.404	0.476	0.761	0.476
		F1	**0.948**	0.923	0.760	0.938	0.900	0.161	0.250	0.160	0.556	0.095
Average	Yes	P	0.995	0.999	0.974	0.93	0.937	0.121	0.237	0.156	0.547	0.053
		R	0.934	0.894	0.628	0.953	0.864	0.29	0.399	0.419	0.79	0.471
		F1	**0.963**	0.944*	0.763*	0.941*	0.899*	0.170*	0.305*	0.218*	0.643*	0.096*

* FlexLog yields a significant higher F1-score than compared baseline.

Table 3.9: Statistics of the sampled subsets of ADFA-U.

$\mathcal{D}_{\#}$	adduser			hydraFTP			hydraSSH			java			web			meter		
	$\mathcal{U}_{\%}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$	$\mathcal{U}_{\#}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$	$\mathcal{U}_{\#}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$	$\mathcal{U}_{\#}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$	$\mathcal{U}_{\#}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$	$\mathcal{U}_{\#}$	$\mathcal{U}^{+}_{\%}$	$\mathcal{U}^{-}_{\%}$
50	1.43	3.88	0.81	1.46	4.37	0.81	1.37	4.46	0.70	1.44	4.08	0.81	1.34	4.05	0.70	1.30	3.77	0.66
500	12.23	32.34	7.19	12.50	36.18	7.23	12.62	36.96	7.30	11.95	32.84	6.98	12.42	34.19	7.19	12.46	31.11	7.66
1000	21.78	53.65	13.80	21.45	57.16	13.53	21.76	58.03	13.82	21.55	54.08	13.84	21.30	53.80	13.49	21.81	52.41	13.91
1500	29.10	67.49	19.47	29.15	70.97	19.82	29.23	72.32	19.79	29.17	70.09	19.47	29.49	69.85	19.79	28.77	67.22	18.80
2000	35.48	78.38	24.72	35.04	82.69	24.41	35.96	82.32	25.81	36.13	79.90	25.71	35.30	79.57	24.66	35.11	76.43	24.43

Figure 3.6 depicts the effectiveness of FlexLog and top baselines trained on the sampled subsets of ADFA-U. Less competitive baselines, such as PLELog, LogAnomaly, DeepLog, LogCluster, and PCA, are not illustrated in the figure for brevity. The dashed horizontal line in each figure represents the state-of-the-art results on each dataset, produced by LightAD trained with full datasets.

Table 3.10 highlights the improvement of FlexLog by calculating F1 score differences in percentage points between FlexLog and baselines. We also report Mann-Whitney U Test results for F1 comparisons across datasets. A "$*$" symbol is appended to the average F1 score if FlexLog significantly outperforms the baseline in terms of F1 score.

Overall, FlexLog consistently achieves superior performance in terms of average F1 scores compared to the baselines when trained with the same amount of labeled data. As

shown in Table 3.10, the average F1 score improvements over the strongest baseline (i.e., FlexLog minus LightAD) for $\mathcal{D}_{\#} = 50$, 500, 1000, 1500, and 2000 are 2 pp, 13 pp, 10 pp, 8 pp, and 7 pp, respectively. Mann-Whitney U tests show the significance of all the improvements, except for the extreme data scarcity scenario with $\mathcal{D}_{\#} = 50$. Notably, even when compared to LightAD trained on the full dataset (represented by the dashed horizontal lines in all plots in Figure 3.6), FlexLog achieves higher or comparable performance with significantly less labeled data. In the remainder of this section, we elaborate on the performance of FlexLog and baselines under different data scarcity scenarios.

Under extreme data scarcity ($\mathcal{D}_{\#} = 50$), we observe that all approaches exhibit poor performance in terms of F1 score, which are lower than 0.60, respectively, across all six configurations of ADFA-U. This limitation likely stems from insufficient training data diversity. As reported in the first row of Table 3.9, the 50 samples selected for each configuration of ADFA-U contain less than 1.46 % of total unique log sequences in full datasets. The percentages of unique anomalous log sequences are also low, with a maximum of 4.46 % ($\mathcal{D}_{\#} = 50$, the *java configuration*). Such low diversity and small size of the labeled datasets tend to be insufficient for the training of all approaches, including FlexLog and baselines.

Under less severe data scarcity ($\mathcal{D}_{\#} = 500, 1000, 1500, 2000$), all methods exhibit improved effectiveness compared to the extreme label scarcity scenario ($\mathcal{D}_{\#} = 50$). FlexLog outperforms all baselines across all six configurations in terms of F1 score. As reported in Table 3.10, the maximum percentage points against LightAD are observed at $\mathcal{D}_{\#} = 500$, reaching 13 pp on average. The average difference values diminish to 7 pp as data size increases to 2000, underscoring FlexLog's advantage with limited labeled data. Mann-Whitney U tests confirm the significance of all the differences ($\mathcal{D}_{\#} = 500, 1000, 1500, 2000$).

The answer to RQ2 is that FlexLog outperforms baselines in F1 scores under **varying data scarcity levels** of ADFA-U except at $\mathcal{D}_{\#} = 50$, where all methods perform poorly due to insufficient data.

3.5.3 RQ3: Time and Memory Efficiency

3.5.3.1 Time Efficiency

Table 3.11 reports the training and inference time of FlexLog and the baselines across different datasets. Column "M" denotes the metric, where "T" represents the training time of full datasets (in seconds) and "I" represents the average inference time of one log sequence (in seconds). Specifically, we include FlexLog's base models—one LLM (Mistral) and three ML models (KNN, DT, and SLFN), which are introduced in § 3.4.3.2. The reported training and inference times represent the total time aggregated across these models (without considering parallel computation).

FlexLog requires substantially more training time than the baselines, with a maximum of 23 706 seconds (6 hours and 35 minutes) on the LOGEVOL-U Spark and SYNEVOL-U datasets. In contrast, the maximum training time of the baselines is recorded as only 1 550 seconds when training NeuralLog on the LOGEVOL-U Hadoop dataset. However, since training is a one-time cost, such a long training time is generally acceptable.

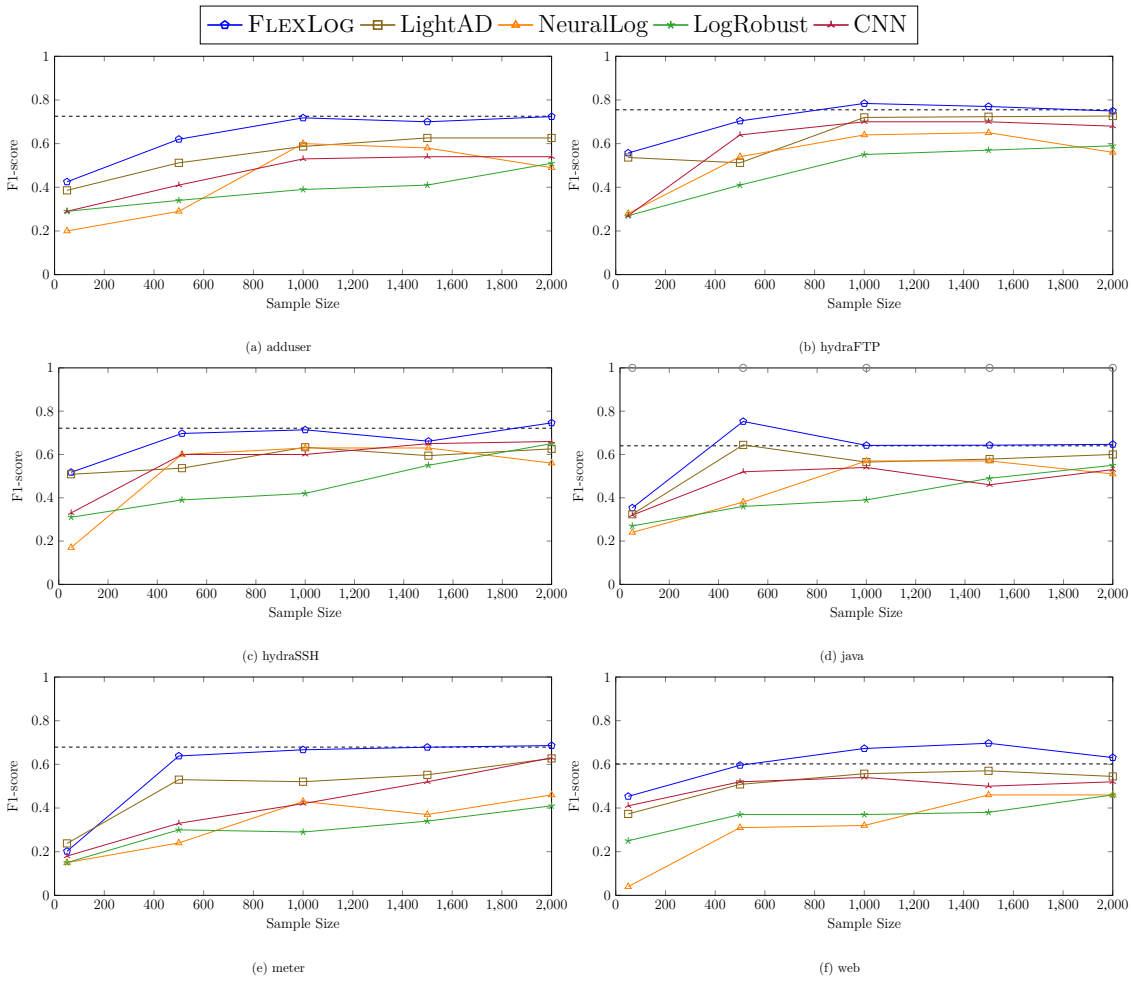


Figure 3.6: Effectiveness of FlexLog and top four baselines trained on varying data scarcity level on ADFA-U.

Table 3.10: F1 score differences (in percentage points) and statistical testing results when comparing FlexLog to baseline methods on the ADFA-U dataset.

Configuration	$\mathcal{D}_{\#}$	**Supervised**				**Semi-S**	**Unsupervised**			
		LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
adduser	50	4	22	13	13	3	10	10	23	19
	500	11	33	28	21	24	36	44	42	55
	1000	13	11	32	18	22	40	45	49	64
	1500	7	12	29	16	40	47	47	47	64
	2000	10	23	21	18	23	47	50	48	67
hydraFTP	50	2	27	28	20	20	22	22	26	38
	500	19	16	29	6	48	33	35	38	58
	1000	6	14	23	8	32	42	42	44	64
	1500	5	12	20	7	47	38	39	42	63
	2000	3	19	16	7	28	35	34	37	64
hydraSSH	50	1	35	21	19	6	17	19	22	28
	500	16	10	31	10	40	32	29	36	52
	1000	8	8	29	11	56	37	31	36	53
	1500	7	4	12	2	35	31	30	30	52
	2000	12	18	9	8	33	34	31	37	57
java	50	3	11	8	3	15	5	1	11	7
	500	11	37	39	23	51	41	47	48	48
	1000	6	7	25	10	50	29	35	36	47
	1500	6	7	15	18	46	29	35	35	48
	2000	5	14	10	12	33	29	36	36	51
meter	50	−4	5	5	2	7	0	1	5	10
	500	11	39	33	30	51	45	46	47	53
	1000	15	23	37	24	58	50	52	49	65
	1500	13	30	33	15	53	50	52	48	63
	2000	6	22	27	5	53	48	46	46	67
web	50	8	41	20	4	14	13	13	22	20
	500	9	29	23	8	17	37	36	34	47
	1000	12	35	30	13	35	40	40	41	60
	1500	13	24	32	20	45	43	42	44	58
	2000	9	17	17	11	42	32	31	37	53
average	50	2	23*	18*	10*	11*	11*	11*	18*	20*
	500	13*	27	30	16*	38*	37*	39*	41*	52*
	1000	10*	16*	29*	14*	42*	40*	41*	42*	59*
	1500	8*	15*	23*	13*	44*	40*	41*	41*	58*
	2000	7*	19*	17*	10*	35*	37*	38*	40*	60*

* FlexLog yields a significant higher F1-score than compared baseline.

FlexLog's average inference time, per log sequence, remains below 1 second, ranging from 0.771 on SynHDFS-U to 0.988 on SYNEVOL-U. While this is significantly higher than traditional ML and DL baselines—some of which make one prediction in milliseconds—it reflects the inherent trade-off between model complexity and efficiency. Simpler models, such as PCA and LogCluster, achieve near-instantaneous inference but suffer from low detection effectiveness, failing to detect critical anomalies (§ 3.5.1). The inference time of FlexLog is acceptable in user-oriented monitoring systems, where inference times on the order of seconds are generally tolerable. Examples include cloud service monitoring (e.g.,

Table 3.11: Training (T) and Inference (I) time of FlexLog and the baselines (in seconds) on ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U.

Config	M	Supervised									Semi-S	Unsupervised			
		FlexLog	Mistral	KNN	DT	SLFN	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
ADFA-U															
adduser	T	16 161	16 160	2	<0.001	<0.001	5	922	221	271	285	276	184	4	0.017
	I	0.836	<0.001	<0.001	0.005	0.842	0.012	0.664	0.234	0.164	0.211	0.025	0.012	<0.001	≪ 0.001
hydraFTP	T	15 906	15 906	1	<0.001	<0.001	4	920	220	269	285	276	181	4	0.017
	I	0.818	0.813	<0.001	<0.001	0.005	0.014	0.635	0.229	0.165	0.211	0.024	0.011	<0.001	≪ 0.001
hydraSSH	T	14 147	14 146	1	<0.001	<0.001	5	923	220	269	285	275	183	4	0.017
	I	0.832	0.832	<0.001	<0.001	0.004	0.014	0.655	0.232	0.164	0.211	0.025	0.012	<0.001	≪ 0.001
java	T	13 933	13 932	1	<0.001	<0.001	6	921	220	270	285	275	180	4	0.017
	I	0.875	0.870	<0.001	<0.001	0.005	0.014	0.673	0.236	0.164	0.211	0.025	0.011	<0.001	≪ 0.001
web	T	14 079	14 078	1	<0.001	<0.001	5	918	220	270	285	276	181	4	0.017
	I	0.864	0.860	<0.001	<0.001	0.004	0.013	0.679	0.231	0.165	0.211	0.025	0.012	<0.001	≪ 0.001
meter	T	15 324	15 323	1	<0.001	<0.001	3	921	220	269	285	276	182	4	0.017
	I	0.888	0.885	<0.001	<0.001	0.003	0.014	0.682	0.238	0.165	0.211	0.025	0.012	<0.001	≪ 0.001
LOGEVOL-U															
Hadoop	T	4 031	4 031	6	<0.001	<0.001	30	1 550	178	316	48	1 340	1 020	21	0.180
	I	0.435	0.414	0.003	0.001	0.017	0.067	0.275	0.078	0.077	0.072	0.004	0.001	<0.001	≪ 0.001
Spark	T	23 706	23 706	*N/A*	≪0.001	≪0.001	0.2	712	232	136	220	944	514	9	0.015
	I	0.844	0.838	*N/A*	<0.001	0.006	0.038	0.564	0.082	0.072	0.006	0.007	0.003	< 0.001	≪ 0.001
SynHDFS-U															
average	T	13 587	13 583	4	<0.001	<0.001	22	1 260	293	355	42	976	1 110	20	0.067
	I	0.771	0.771	≪0.001	≪0.001	<0.001	0.005	0.259	0.008	0.016	0.007	0.004	0.002	< 0.001	≪ 0.001
SYNEVOL-U															
average	T	23 706	23 706	*N/A*	<0.001	<0.001	0.2	712	232	136	220	944	514	9	0.015
	I	0.988	0.979	*N/A*	<0.001	0.009	0.038	0.703	0.116	0.076	0.008	0.013	0.004	< 0.001	≪ 0.001

AWS CloudWatch [149], Microsoft Azure Monitor [150]), where system logs are analyzed to detect performance issues; IT infrastructure monitoring (e.g., Prometheus [151]), where server and network health metrics are updated periodically; and industrial IoT monitoring, where manufacturing systems and smart grids detect equipment failures. In these cases, inference time or response time of an anomaly detector within seconds still allows for timely interventions. In contrast, latency-sensitive domains, such as high-frequency trading systems, demand millisecond-level inference time, as decisions must be made within microseconds to capitalize on market fluctuation [152]. In such scenarios, the inference time of FlexLog, which is on the order of seconds, is not acceptable.

We remark that although FlexLog is not as efficient as traditional methods, its superior effectiveness justifies its application in scenarios where reliability is paramount. In anomaly detection, false negatives—undetected anomalies—can have far greater consequences than minor delays in inference time. For instance, in high-stakes environments like cybersecurity or critical infrastructure monitoring, failing to detect an anomaly or an intrusion attempt could lead to data breaches, financial losses, or system-wide failures. While simpler models offer faster inference, they often sacrifice effectiveness, leading to unacceptable numbers of false positives and false negatives.

To further explain the time cost of FlexLog, Table 3.11 shows the training and inference time of its base models, which include one LLM (Mistral) and three ML models (KNN, DT, and SLFN). We find that Mistral requires substantially more training time and average inference time per sequence compared to the three ML base models, accounting for the majority of time needed for FlexLog. Specifically, while Mistral's training and

inference time remain under 7 hours and 1 s, respectively, all ML base models take less than 6 s for training and less than 0.001 s for inference per log sequence. This discrepancy is primarily due to Mistral's significantly large number of trainable parameters. As a result, Mistral has the most impact on the time efficiency of FlexLog compared to ML models. However, with parallel computing and more powerful hardware, the efficiency of LLMs can easily be significantly improved, thereby enhancing the overall efficiency of FlexLog.

3.5.3.2 Memory Efficiency of FlexLog's Cache Mechanism

FlexLog leverages a caching mechanism to improve efficiency during inference by storing predictions for previous log sequences, as described in § 3.3.2. The cache improves FlexLog's time efficiency by reducing repetitive computation, although it introduces extra memory overhead. To evaluate the memory efficiency of this caching component, we report the memory usage, defined in § 3.4.2.4, during inference across our datasets. For ADFA-U, which contains the longest sequences with an average length of 461 templates, FlexLog consumes between 17.16 MB and 19.60 MB of memory. For LOGEVOL-U, which has the largest number of log sequences, the memory usage remains below 4 MB—specifically, 1.75 MB for Hadoop and 3.5 MB for Spark. For our synthetic datasets, memory usage also stays below 1 MB, averaging 576 kB and 2.88 MB for SynHDFS-U and SYNEVOL-U, respectively, across different injection ratios. Overall, this level of memory usage demonstrates the scalability potential of FlexLog 's caching mechanism, especially when balanced against the time savings presented as part of the answer to RQ4 (§ 3.5.4.1). Regarding scalability in systems with extremely long or highly diverse sequences, we note that the cache's *delete* function helps keep memory consumption under control (see § 3.3.2 for details). Further analysis of memory efficiency will depend on the specific memory resources available in the monitoring system during inference.

The answer to RQ3 is that while FlexLog is **not the most time-efficient** in ULAD inference, it processes each log sequence within **1** s **on average**. FlexLog's cache memory remains **below 4** MB for most datasets (up to 19.6 MB for ADFA-U), confirming its **memory efficiency**.

3.5.4 RQ4: Configuration Impact

In this section, we investigate the impact of FlexLog's different ablation configurations, such as excluding the cache mechanism (§ 3.5.4.1), excluding RAG (§ 3.5.4.2) and different choices of base models in ensemble learning (§ 3.5.4.3). Additionally, we investigate the impact of alternative LLMs in the configuration of FlexLog (§ 3.5.4.3) to highlight the advantages of using Mistral Small as the LLM component.

3.5.4.1 Impact of Cache-empowered Inference

FlexLog maintains a cache C to avoid redundant predictions incurred by recurring log sequences, thus improving inference efficiency. To assess the impact of C on inference

Table 3.12: FlexLog vs. FlexLog w/o cache — Comparisons of inference time per log sequence (in seconds) on ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U

Config	ADFA-U							LOGEVOL-U			SYNEVOL-U	SynHDFS-U
	adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
FlexLog	0.842	0.818	0.832	0.875	0.864	0.888	**0.853**	0.435	0.844	**0.628**	**0.941**	**0.771**
FlexLog w/o cache	0.896	0.861	0.898	0.916	0.909	0.952	0.905	0.793	1.086	0.940	0.988	0.794
Difference (s)	-0.054	-0.043	-0.066	-0.041	-0.045	-0.064	-0.052*	-0.358	-0.242	-0.312*	-0.047*	-0.023

* FlexLog yields a significant lower inference time than FlexLog without cache.

time, we compare FlexLog and FlexLog without cache (denoted by "w/o cache") in terms of inference time across four datasets: ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U. The results of this comparison are reported in Table 3.12. Column "Config" indicates the subjects under comparison — FlexLog and FlexLog w/o cache; the following four columns "ADFA-U", "LOGEVOL-U", "SYNEVOL-U" and "SynHDFS-U" report the results of inference time per log sequence (in seconds) for each respective dataset. For the real-world datasets (ADFA-U and LOGEVOL-U), we report inference time for each configuration. In contrast, for the synthetic datasets (SYNEVOL-U and SynHDFS-U), we provide only the average inference time across configurations. This is because different configurations of SYNEVOL-U and SynHDFS-U vary in the injection ratio of changes, which has minimal impact on inference time, leading to similar inference time across configurations. Hence, we report only the average inference time on synthesized datasets for brevity. The last row "Difference" indicates the difference between the inference time of FlexLog and that for FlexLog w/o cache. Similar to RQ1 (§ 3.5.1), we conducted Mann-Whitney U tests on each dataset to assess the statistical significance of the differences in inference times; the symbol "*" indicates the inference time of FlexLog is significantly lower than that of FlexLog w/o cache.

We observe that the introduction of the cache consistently reduces the inference time across all datasets. The reduction on two real-world datasets—ADFA-U and LOGEVOL-U—and SYNEVOL-U are more pronounced than SynHDFS-U, with reductions of 0.052, 0.312, and 0.047, respectively. Mann-Whitney U tests confirm these reductions are statistically significant, whereas the reduction on SynHDFS-U is smaller (0.023) and statistically not significant. However, as discussed in § 3.5.3, the practical impact of caching is limited in user-oriented monitoring systems, where inference times below 1 second are generally considered acceptable. While caching improves efficiency, the reduction in inference time might be too small to make a noticeable difference in such systems.

That said, we remark that the impact of caching previously seen log sequences depends on how frequently identical log sequences reappear in real-world systems. In our experiments, 6 %, 43 %, 2.7 %, and 2.2 % of log sequences appear more than once in the testing dataset of ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U, respectively. Hence, the average reduction of inference time on each dataset, ordered from the greatest to the least, is as follows: LOGEVOL-U (−0.312 seconds), ADFA-U (−0.052 seconds), SYNEVOL-U (−0.047 seconds), and SynHDFS-U (−0.023 seconds). In practice, in many operational environments, such as distributed cloud computing frameworks (e.g., Hadoop [153] and Spark [154]) and security monitoring systems, certain types of log sequences, such as scheduled job reports and system diagnostics, occur repeatedly over time.

Table 3.13: FlexLog vs. FlexLog w/o RAG — Comparisons in terms of F1 score (in percentage points) on the ADFA-U dataset.

Config	adduser	hydraFTP	hydraSSH	java	meter	web	Average
FlexLog	71.8	78.4	72.3	64.2	68.2	67.2	**70.4**
FlexLog w/o RAG	68.8	70.5	64.8	62.2	65.9	64.1	66.0
Difference (pp)	3.0	7.9	7.5	2.0	2.3	3.1	4.4*

* FlexLog yields a significant higher F1-score than FlexLog without RAG.

In these cases, the cache mechanism of FlexLog can significantly improve inference efficiency, but its impact depends on the extent of redundant computations and the acceptable latency requirements of the system.

3.5.4.2 Impact of RAG

FlexLog employs RAG to retrieve context for log sequences, providing relevant information that enriches the prompts. In our experiments, only ADFA-U has available contextual information, specifically Linux system call descriptions. Hence, RAG is only applied in the experiments on the ADFA-U dataset. To assess the impact of RAG, we compare the F1 scores of FlexLog and FlexLog without RAG (denoted by "w/o RAG") on ADFA-U. The results of this comparison are reported in Table 3.13. Column "Config" represents the subjects under comparison — FlexLog and FlexLog w/o RAG. Column "adduser", "hydraFP", "hydraSSH", "java", "meter", and "web" reports the F1 scores of six different configurations of ADFA-U; configuration details are described in § 3.4.2.2. The last column "Average" indicates the average F1 score across all six configurations; the last row "Difference (pp)" reports the difference between the F1 scores of FlexLog and those of FlexLog w/o RAG. We performed a Mann-Whitney U test to assess the significance of the differences; however, testing on individual configurations was not feasible since we repeated the experiments on each configuration 5 times, which does not provide sufficient statistical power. Instead, a Mann-Whitney U test was run across all six configurations, increasing the test sample size to 30.

The results in Table 3.13 show that FlexLog outperforms FlexLog w/o RAG on all six configurations of ADFA-U, with differences ranging from 2 pp to 7.9 pp. On average, RAG improves the F1 score of ULAD from 0.660 to 0.704 (4.4 pp). A Mann-Whitney U test confirms this improvement is statistically significant. Notably, across all six configurations of ADFA-U—each representing distinct unseen attack types in the testing set—FlexLog consistently achieved higher effectiveness when RAG was applied. The RAG component enhances the FlexLog's capacity to generalize to unseen failure and attack patterns by dynamically retrieving relevant contextual knowledge during inference. While the magnitude of improvement may vary across datasets depending on the availability and quality of contextual information, these findings suggest that RAG consistently improves effectiveness. Overall, FlexLog does not assume a fixed or progressive change model; instead, RAG enables adaptive context integration, improving robustness to unseen or evolving patterns.

These results confirm that the RAG component of FlexLog is effective in improving F1 scores on ADFA-U. This suggests that RAG could similarly improve performance on other datasets that have contextually rich log information. For example, in cloud computing platforms (e.g., AWS or Azure), where logs often include metadata such as instance IDs, resource types, or service configurations, RAG could enhance the effectiveness of ULAD by providing context that helps identify patterns or anomalies related to specific resources or services.

3.5.4.3 Impact of Base Model Choices in Ensemble Learning

FlexLog employs an ensemble of four base models for ULAD, leveraging diverse perspectives to enhance predictive effectiveness. As described in § 3.4.3, the ensemble in FlexLog includes one LLM (Mistral) and three ML models (KNN, DT, and SLFN), which were selected empirically due to their superior performance across different datasets. In this section, we present the results of different base model choices on ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U, including ablation studies (e.g., removing individual base models and removing all ML base models) and replacing Mistral with alternative LLMs (Llama 3.1 and GPT-4o).

Ablation Study Table 3.14 reports ablation studies of FlexLog on all ULAD configurations. Column "Config" denotes the configuration of Flexlog; for instance, "w/o SLFP" represents FlexLog without SLFP base model, and "w/o ML" represents FlexLog without the three ML models (KNN, DT, and SLFN), resulting in standalone Mistral. To evaluate the individual contribution of each base model, we first assessed four configurations of FlexLog, each obtained by removing a single base model: FlexLog w/o Mistral, w/o KNN, w/o SLFN, and w/o DT. The results in Table 3.14 indicate that removing Mistral leads to the most significant drop in F1 scores, with a maximum reduction of 6.7 pp on ADFA-U adduser. Removing any of the ML models (KNN, DT, or SLFN) also results in reduced F1 scores across all configurations in all the datasets.To assess the statistical significance of these reductions, we conducted Mann-Whitney U tests on each dataset. The results confirm that the decreases in F1 scores are statistically significant in all cases, except for FlexLog w/o KNN, w/o DT, and w/o SLFN on LOGEVOL-U. This highlights the effectiveness of each ML base model in contributing to FlexLog 's overall effectiveness. Further, we assess the contribution of all ML models by excluding all three ML models from FlexLog, leaving only Mistral for predictions. This resulted in decreased F1 scores across most datasets, except for LOGEVOL-U Spark and SYNEVOL-U, where "w/o ML" outperformed FlexLog by 7 pp (96.2% − 89.2%) and 0.8 pp (97.9% − 97.1%), respectively. Mann-Whitney U tests show that FlexLog significantly outperforms "w/o ML" in terms of F1 score on ADFA-U and SynHDFS-U. The F1 score difference on SYNEVOL-U between FlexLog and "w/o ML" is statistically not significant, whereas on LOGEVOL-U Spark, "w/o ML" achieves a significantly higher F1 score than FlexLog. These results align with the findings from removing individual ML base models, further suggesting that the ML base models contribute negatively to FlexLog's performance on LOGEVOL-U Spark and , leading to "w/o ML" outperforming the ensemble.

A likely explanation for the better performance of FlexLog without any ML base

models on LOGEVOL-U Spark and SYNEVOL-U is these datasets' extreme class imbalance. As detailed in § 3.4.2.1, LOGEVOL-U Spark and SYNEVOL-U share the same training dataset, sampled from LogEvol Spark 2, which is highly imbalanced: only 16 % of log sequences in the sampled training dataset are anomalous, compared to 50 % in ADFA-U and LOGEVOL-U Hadoop, and 57 % in SynHDFS-U. It is well known that traditional ML models, such as KNN, SLFN, and DT, struggle with highly imbalanced datasets [155]. Since FlexLog integrates these ML models, its performance is negatively affected.

To mitigate data imbalance, we experimented with both down-sampling and over-sampling techniques [156]. Down-sampling by reducing normal logs to match the number of anomalous logs led to a decrease of 5 pp in F1 score. For over-sampling, we applied several standard strategies, including duplicating anomalous logs, adding small perturbations to representation vectors, and using the Synthetic Minority Oversampling Technique (SMOTE) [157] on representation vectors. However, none of these approaches improved the effectiveness of the ML models. The key challenge lies in the representation of log sequences as count vectors. Unlike continuous feature spaces where interpolation can generate plausible synthetic samples, count vector representations encode categorical relationships, making common over-sampling techniques such as SMOTE [157] prone to producing unrealistic or noisy data. In contrast, standalone Mistral ("w/o ML") remains robust in predictive effectiveness, leveraging its pretraining on vast and diverse corpora to mitigate data imbalance. Based on these findings, we *recommend* using standalone Mistral instead of the full FlexLog when dealing with extremely imbalanced training datasets, eliminating the negative effect of poorly trained ML models.

For future improvements, more advanced data augmentation techniques, such as Generative Adversarial Networks (GANs), could be used to generate synthetic log sequences directly, rather than modifying their count vector representations. This avoids the issue of unrealistic or noisy data caused by over-sampling in a discrete feature space, making the synthetic data more useful for training ML models on imbalanced datasets like LOGEVOL-U Spark. As a result, this could potentially improve FlexLog 's overall performance.

Alternative LLMs. Table 3.15 reports the F1 score of FlexLog with three LLM choices. Column "Config" represents different configurations of FlexLog, wherein Mistral is replaced by Llama 3.1 8B ("*Mistral → Llama*") and GPT-4o ("*Mistral → GPT*"). Column "Source" indicates whether the employed LLM is open-source or closed-source. Experimental results indicate that GPT-4o and Mistral achieve comparable effectiveness, both consistently outperforming Llama across all configurations and datasets. Mann-Whitney U tests at the dataset level confirm that the F1 score differences between FlexLog and " *Mistral → GPT*" are not significant across all four datasets. However, FlexLog significantly outperforms "*Mistral → Llama*" on ADFA-U, LOGEVOL-U, and SynHDFS-U, while the difference on SYNEVOL-U is not statistically significant. These findings underscore the effectiveness of Mistral for ULAD, demonstrating performance on par with the closed-source GPT-4o while avoiding the financial cost associated with API invoking. Thus, we recommend Mistral as a cost-effective yet competitive base model for FlexLog.

Table 3.14: Ablation studies of FlexLog on all unstable datasets.

Config	ADFA-U							LOGEVOL-U			SynHDFS-U	SYNEVOL-U
	adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
FlexLog	**0.718**	**0.784**	**0.723**	**0.642**	**0.682**	**0.672**	**0.704**	0.982	0.892	0.937	**0.972**	0.971
w/o Mistral	0.645	0.769	0.692	0.628	0.639	0.616	0.664*	0.975	0.841	0.908*	0.939*	0.936*
w/o KNN	0.683	0.768	0.654	0.621	0.651	0.579	0.659*	0.980	*N/A*	0.980	0.945*	*N/A*
w/o DT	0.667	0.749	0.690	0.640	0.654	0.623	0.670*	0.973	0.875	0.924	0.948*	0.959*
w/o SLFN	0.677	0.691	0.613	0.641	0.647	0.556	0.637*	0.978	0.871	0.924	0.934*	0.948*
w/o ML	0.579	0.591	0.630	0.628	0.674	0.571	0.612*	**0.998**	**0.962**	**0.980**†	0.928*	**0.979**

* FlexLog yields a significant higher F1-score than the ablation configuration.
† FlexLog yields a significant lower F1-score than the ablation configuration.
N/A Not applicable. KNN is excluded from FlexLog on the SynHDFS-U and LOGEVOL-U datasets (see § 3.4.3.2)

Table 3.15: F1 scores of using alternative LLMs in FlexLog.

Config	Source	ADFA-U							LOGEVOL-U			SynHDFS-U	SYNEVOL-U
		adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
FlexLog	open	0.718	0.784	**0.723**	0.642	0.682	0.672	0.704	0.982	0.892	**0.937**	0.972	**0.971**
Mistral → Llama	open	0.679	0.743	0.707	0.593	0.669	0.591	0.664*	0.941	0.850	0.895*	0.949*	0.970
Mistral → GPT	closed	**0.721**	**0.786**	0.718	**0.648**	**0.690**	**0.696**	**0.710**	**0.981**	0.886	0.933	**0.976**	**0.971**

* FlexLog yields a significant higher F1 score compared to using the alternative LLM.

The answer to RQ4 is that the full FlexLog configuration—combining **cache**, **RAG**, and an ensemble of **KNN, DT, SLFN, and Mistral**—performs best, with each component improving efficiency or effectiveness. Among LLMs, **Mistral** is a cost-effective choice, matching the performance of GPT-4o while outperforming Llama.

3.5.5 Discussion

In this section, to guide AIOps engineers, we highlight the implications of our study for ULAD.

3.5.5.1 Token Consumption of LLMs

In the early times of leveraging attentive language models, language models accepted limited input tokens, such as a maximum of 512 input tokens for BERT [158], making it challenging to apply them to long log sequences. However, as language models expanded into LLMs, the maximum input token limit also increased, both for open-source and closed-source models. Table 3.16 reports the input token limits of the LLMs used in our work and compares them with the maximum token consumption observed for each dataset. Column "Model" denotes LLMs used in our experiments. Column "Source" indicates whether the LLM is open-sourced or closed-source. Column "Input Token Limit" reports the input token limit specific to each LLM. Column "Maximum Input Token" denotes the maximum token consumption of FlexLog's prompts on each dataset. We note that these values vary across LLMs due to variations in their tokenization processes. Our results show that FlexLog's token consumption remains well within input token limits for all LLMs, indicating that the challenge of input token limits has been alleviated and even eliminated

for our datasets. The highest usage is only 37.7 % (24 735 out of the maximum limit of 65 536 tokens), with prompts used by GPT-4o for ADFA-U. Notably, in our experiments, we have included various log datasets with log sequences as long as 4 474 templates (see Table 3.1), and yet LLMs effectively accommodate them. Moreover, recent advances in LLMs can potentially extend the token limit to one million tokens [159, 160], making it no longer a hard limitation for using LLMs.

Table 3.16: Overview of token consumption of ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U on open-source and closed-source LLMs

Model	**Accessibility**	**Input Token Limit**	**Maximum Input Token**			
			ADFA-U	**LOGEVOL-U**	**SynHDFS-U**	**SYNEVOL-U**
Mistral Small	open	128 000	25 848	28 934	1 243	16 840
Llama 3.1 8B	open	128 000	21 371	24 798	862	12 533
GPT-4o	closed	65 536	21 383	24 735	860	12 479

3.5.5.2 Error Analysis

In RQ4, we observed that removing any base model from FlexLog generally decreases effectiveness. To better understand the role of the LLM base model (i.e., Mistral) and ML base models (i.e., KNN, DT, and SLFN), in this section, we analyze the log sequences that FlexLog correctly classify but mis-classify when either Mistral or ML models are removed. We denote these mis-classified log sequences as $\texttt{MIS}_{w/oMistral}$ and $\texttt{MIS}_{w/oML}$, respectively. We investigate, for each dataset, whether $\texttt{MIS}_{w/oMistral}$ and $\texttt{MIS}_{w/oML}$ differ in various characteristics to highlight the unique contribution of LLM and ML to the effectiveness of FlexLog. Specifically, we focus on the unseen log templates in $\texttt{MIS}_{w/oMistral}$ and $\texttt{MIS}_{w/oML}$ because unseen log templates pose a key challenge in ULAD. Traditional anomaly detectors often rely on predefined patterns and struggle with generalization; in contrast, FlexLog, by means of the integration of LLM, may be better at detecting anomalies involving novel log templates. By analyzing the misclassified cases in $\texttt{MIS}_{w/oMistral}$ and $\texttt{MIS}_{w/oML}$, we can determine whether the LLM (Mistral) or ML models (KNN, DT, and SLFN) contribute more to handling novel log templates, helping us understand their complementary strengths in FlexLog.

Table 3.17 reports the error analysis results across all four datasets: ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U. Column "Data" specifies the dataset; Column "Config" reports the configuration of each dataset; Column "Condition" shows the subject of the statistics, including the testing dataset, $\texttt{MIS}_{w/oMistral}$, and $\texttt{MIS}_{w/oML}$. Column "# Sequences" reports the number of sequences; column "Sequence Length" indicates the average, minimum, and maximum length of the sequences, denoted as "avg", "min", and "max", respectively. Column "% Anomaly" reports the percentage of anomalous log sequences, and column "% Unseen Template" denotes the percentage of log sequences that have at least one unseen log template. Overall, for each dataset, $\texttt{MIS}_{w/oMistral}$ and $\texttt{MIS}_{w/oML}$ demonstrate different characteristics in terms of log sequence length, percentage of anomaly, and percentage of unseen templates, highlighting how Flexlog's base models complement each other in an effective prediction.

In terms of unseen templates, in LOGEVOL-U Hadoop and SynHDFS-U, we did not observe any in either $\texttt{MIS}_{w/oML}$ or $\texttt{MIS}_{w/oMistral}$. In ADFA-U, $\texttt{MIS}_{w/oMistral}$ contains 7.38 % unseen log templates, much higher than that of $\texttt{MIS}_{w/oML}$ (1.89 %). Similarly, in LOGEVOL-U Spark, the percentage of unseen log templates in $\texttt{MIS}_{w/oMistral}$ reaches 33.33 %, while $\texttt{MIS}_{w/oML}$ contains no unseen template-related mis-classifications. In SYNEVOL-U, 11.76 % of misclassifications in $\texttt{MIS}_{w/oMistral}$ are related to unseen templates, higher than that in $\texttt{MIS}_{w/oML}$ (9.28 %). Overall, we observe a higher percentage of unseen templates in $\texttt{MIS}_{w/oMistral}$ than $\texttt{MIS}_{w/oML}$ across most datasets. The higher percentage of unseen log templates in $\texttt{MIS}_{w/oMistral}$, across several datasets, indicates that Mistral plays a crucial role in handling unseen log templates. This highlights LLM's adaptability in recognizing new templates, making it particularly valuable for ULAD, where logs are unstable due to environment and system evolution. In contrast, perhaps unsurprisingly, ML models appear to rely more on established patterns, struggling with new templates.

Table 3.17: Overview of Error Analysis of ADFA-U, LOGEVOL-U, SYNEVOL-U, and SynHDFS-U when Mistral or ML models are removed from FlexLog

Data	Config	Condition	# Sequence	Sequence Length			% Anomaly	% Unseen Template
				avg	min	max		
ADFA-U	all ULAD	Testing Dataset	5 146	523.98	75	4 494	14.34	3.54
		$\texttt{MIS}_{w/oMistral}$	149	476.74	82	2 513	0	7.38
		$\texttt{MIS}_{w/oML}$	264	610.48	80	3 089	11.74	1.89
LOGEVOL-U	Hadoop ULAD	Testing Dataset	5 329	13.41	1	50	9.8	74.37
		$\texttt{MIS}_{w/oMistral}$	5	41.8	9	50	0.0	0.0
		$\texttt{MIS}_{w/oML}$	0	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
	Spark ULAD	Testing Dataset	4 045	81.70	1	1 977	1.78	38.07
		$\texttt{MIS}_{w/oMistral}$	6	15.33	4	35	100	33.33
		$\texttt{MIS}_{w/oML}$	1	26.00	26	26	100	0
	all ULAD	Testing Dataset	9 374	42.88	1	1 977	6.34	58.70
		$\texttt{MIS}_{w/oMistral}$	11	29.83	4	50	54.54	18.18
		$\texttt{MIS}_{w/oML}$	1	26.00	26	26	100.00	0.00
SynHDFS-U	all ULAD	Testing Dataset	3 744	26.91	10	57	22.22	0.43
		$\texttt{MIS}_{w/oMistral}$	24	40.08	35	48	0.0	0.0
		$\texttt{MIS}_{w/oML}$	92	32.43	19	52	5.43	0.0
SYNEVOL-U	all ULAD	Testing Dataset	15 406	151.07	1	1 022	2.96	91.3
		$\texttt{MIS}_{w/oMistral}$	34	135.03	6	323	100	11.76
		$\texttt{MIS}_{w/oML}$	23	39.04	4	177	0	9.28

N/A Not applicable as no prediction errors were observed in this configuration

3.5.6 Threats to Validity

3.5.6.1 Internal Validity

Data Leakage from LLMs. One potential threat to internal validity is data leakage, where test data may inadvertently overlap with training data. Although we performed deduplication across all test datasets, the risk of data leakage cannot be entirely eliminated,

as some test data may have been included in the pretraining corpus of LLMs. This could lead to inflated effectiveness. To mitigate this risk, we evaluated FlexLog not only on publicly available datasets (ADFA, LOGEVOL, and SYNEVOL-U), but also on the SynHDFS-U dataset, which we synthesized ourselves. In addition, the cut-off date of Mistral Small is August 2023, and LOGEVOL and SYNEVOL-U were introduced after this date. Hence, these datasets cannot be part of its pretraining corpus, ensuring a more reliable assessment of FlexLog 's effectiveness.

Selection of Training Dataset Sizes. Another threat to internal validity arises from the selection of training dataset sizes ($\mathcal{D}_{\#}$) for evaluating data efficiency. Since $\mathcal{D}_{\#}$ directly affects the performance of FlexLog and the baselines, an exhaustive evaluation across all possible values is infeasible. To address this limitation, we systematically experimented with multiple $\mathcal{D}_{\#}$ values, as detailed in § 3.5.2, ranging from 50 to 2 000. This range allows us to provide a comprehensive analysis of data efficiency while remaining within our computational constraints.

Selection of FlexLog's Base Models. Since FlexLog is an ensemble learning-based approach, its effectiveness is significantly influenced by the selection of base models. While it is impractical to evaluate all possible model combinations, we carefully selected representative base models from the literature on log-based anomaly detection, including KNN, DT, SLFN, LightAD [12], NeuralLog [37], CNN [22], LLaMA 3.1, and GPT-4o. Through extensive empirical evaluation, we tested multiple model combinations and ultimately selected KNN, DT, MLP, and Mistral due to their consistently high effectiveness and robustness across both real-world and synthesized datasets.

Synthetic Datasets. The synthetic datasets in our experiments are generated by injecting different levels of instability. One potential threat to internal validity lies in the realism of this injected instability and the validity of labels after injections. To mitigate this threat, we adopted datasets synthesized by experts and applied well-established injection strategies from the literature. Specifically, the SYNEVOL-U dataset, proposed by Huo et al [13], was annotated by two experienced Spark developers and carefully reviewed after annotation. For SynHDFS-U datasets, which we constructed following the SYNEVOL-U approach, we applied only sequence-level changes that are unlikely to affect the overall label of a log sequence. To guide this process, we leveraged the decision tree analysis [97] of the original dataset to identify low-impact templates. These precautions help maintain the reliability of the labels, despite the synthetic nature of the data.

Consistency of Labels. For faster inference, FlexLog's cache mechanism reuses the prediction for previously seen, identical log sequences. However, the assumption that identical log sequences always correspond to the same label may not hold in extreme log evolution cases, particularly in security-critical domains. This assumption is nevertheless valid for all of our datasets, where each unique log sequence is consistently labeled. To further mitigate this risk, the *update* function in FlexLog's cache enables label revisions by system administrators if changes are observed over time.

Cascading Effect of Parsing Errors. The use of log parsing introduces a potential risk

of parsing errors, which could negatively affect the effectiveness of FlexLog. However, prior work by Khan et al. [137] reports no strong correlation between log parsing accuracy and anomaly detection accuracy, suggesting that minor parsing errors do not necessarily degrade the effectiveness of FlexLog. Our evaluation shows that FlexLog, as a parsing-based approach, achieves significantly higher effectiveness than NeuralLog and LightAD, both of which operate on raw logs. Specifically, LightAD uses some of the same base models as FlexLog but applies them without log parsing. In FlexLog, this risk is further mitigated through two design factors. First, we adopt widely used and reliable parsers to reduce the likelihood of significant parsing errors (see § 3.4.3.1). Second, the architecture limits error propagation: predictions are aggregated via majority voting across multiple base models, preventing a single erroneous output from dominating the final decision. Additionally, the fine-tuned LLM in FlexLog leverages contextual information, making it resilient to potential inconsistencies introduced by parsing.

3.5.6.2 Conclusion Validity

It is widely acknowledged that LLMs often produce non-deterministic responses even when provided with identical prompts, posing a potential threat to the conclusion validity of FlexLog. To address this challenge, as detailed in § 3.4.3, we configured the LLMs to generate responses with minimal randomness, achieved by setting the temperature parameter to 0. To further reduce the influence of randomness from our evaluation results, as discussed in § 3.4.3, we ran each experiment configuration five times and calculated the average value as the final result. We also conducted Mann-Whitney U test to assess the significance of effectiveness and efficiency differences between FlexLog and baselines. To ensure sufficient statistical power, we performed statistical testing at the dataset level rather than the configuration level, since each configuration only has five samples. Aggregation at the dataset level ensures each test includes at least 10 samples.

3.5.6.3 External Validity

. A potential threat to external validity is the impact of highly unstable logs on anomaly detection performance. While FlexLog achieves state-of-the-art effectiveness, its performance, like that of other methods, may degrade when log instability is extreme. For example, the introduction of YARN for job management in Hadoop 2 resulted in substantial architectural modifications, leading to significant changes in its logs [161]. Such drastic shifts pose challenges for all existing methods, potentially limiting their applicability in highly unstable logging environments. To address this concern, we evaluate FlexLog and the baselines across diverse datasets that capture two primary sources of log instability: software evolution (LOGEVOL-U) and environment change (ADFA-U). Additionally, we conduct experiments on two synthesized datasets (SynHDFS-U and SYNEVOL-U) that simulate instability levels ranging from 0 % to 30 %. Our findings indicate that while all methods experience performance degradation as instability increases, FlexLog remains robust in terms of precision, recall and F1 score. With a minimum F1 score of 0.948 (30 % template level injections on SYNEVOL-U), FlexLog consistently outperforms all

the baselines, demonstrating greater robustness in handling highly unstable logs. Nevertheless, further evaluation on a broader range of real-world systems is necessary to fully assess the limitations of its applicability.

3.6 Related Work

3.6.1 Anomaly Detection on Unstable Logs

Log-based anomaly detection has been extensively studied in the literature to enhance the dependability of software-intensive systems [12, 110, 162]. However, only a few studies have investigated anomaly detection on unstable logs [13, 14, 38, 59], a common situation in practice. Zhang et al. [14] first identified such a challenge and proposed LogRobust (see § 3.4.2.3) to leverage an attention-based Bi-LSTM as an anomaly detector. They also created a new unstable log dataset called Synthetic HDFS to evaluate the effectiveness and robustness of LogRobust. This inspired a number of follow-up works, including supervised [38, 59] and unsupervised approaches [13].

Supervised approaches like HitAnomaly [38] and SwissLog [59] require training with a labeled dataset, encompassing both normal and anomalous data. SwissLog adopts the same architecture (i.e., Bi-LSTM) as LogRobust and aims to further improve it by incorporating time embeddings and Bert-based semantic embeddings. HitAnomaly, however, leverages a much larger model based on a hierarchical transformer architecture. The high complexity of the HitAnomaly model allows it to tackle not only the static parts of log messages but also dynamic parts, such as numerical values that have been masked in the log templates. Experiment results demonstrate the superiority of HitAnomaly on stable logs compared to LogRobust, while showing a robust performance for small injection ratios (under 20%) in unstable logs and being outperformed by LogRobust from 20% to 30%.

Huo et al. [13] proposed EvLog, an unsupervised approach leveraging a multi-level semantics extractor and attention mechanism to identify anomalous log messages in unstable logs. Unlike FlexLog, EvLog is a parser-free method to combat potential parsing errors in a dataset. However, by avoiding log parsing, EvLog may face challenges in generalizing to datasets with diverse or domain-specific log formats, as it relies solely on semantic extraction without leveraging structured context. As part of the EvLog study, they also introduced two log datasets: LOGEVOL and SYNEVOL (referred to as SYNEVOL-U in this chapter), which serve as valuable benchmarks for ULAD research, including our work.

To summarize, compared to existing ULAD approaches, FlexLog: 1) leverages the synergy of ML models and LLMs through ensemble learning 2) requires significantly less training data, reduces labeling cost, 3) achieves state-of-the-art effectiveness across all datasets, outperforming baselines in terms of F1 scores consistently while being trained on limited labeled data.

3.6.2 Application of LLMs to Log Analysis

Over the past few years, LLMs have been widely adopted on different log-related software engineering tasks to enhance effectiveness and generalizability, including anomaly detection and log parsing [100, 136, 163].

Anomaly detection. The application of LLMs in the field of anomaly detection started by leveraging BERT [158] to capture contextual information of logs with semantic-based representations. LogBERT [10] utilizes BERT to learn the semantics of normal log messages and predicts an anomaly where the representation of log messages of an input sequence deviates from the distribution of normal log sequences. Le and Zhang [37] proposed NeuralLog (discussed in § 3.4.2.3).

Han et al. [164] introduced LogGPT, which leverages reinforcement learning to fine-tune GPT-2 for anomaly detection. More recently, Liu et al. [100] proposed LogPrompt, which adopts LLMs such as GPT-3 and Vicuna [165] for online log parsing and anomaly detection via in-context learning; we discussed the reasons for not considering LogPrompt as a baseline in § 3.4.2.3. He et al. [99] proposed LLMeLog, which leverages a BERT model fine-tuned with log sequences enriched with contextual information that was retrieved by GPT-3.5. Inspired by LLMeLog, we further explored RAG with various LLMs, including closed-source and open-source LLMs. Additionally, we equipped LLMs with a cache mechanism and ensemble learning to enhance their effectiveness and data efficiency. Note that we did not consider LLMeLog as a baseline in this work due to the unavailability of their replication package.

Log Parsing. Le and Zhang [105] explored the in-context learning of ChatGPT [114] on log parsing and achieved promising results with zero-shot and few-shot prompts. Xu et al. [163] proposed DivLog, another few-shot, in-context learning method that constructs prompts with five labeled examples for each target log template. DivLog explicitly optimizes the diversity of included examples using the Determinantal Point Process (DPP) [166], reducing the potential biases in the examples by maximizing sample diversity. Jiang et al. [103] proposed a novel parser called LILAC, equipping LLMs with an adaptive cache to reduce the LLM query times and, consequently, the efficiency. Recently, Pei et al. [167] introduced a self-evolutionary LLM-based parser, which identifies new log templates by grouping history log messages. Fewer studies focus on fine-tuning LLMs for log parsing. Le et al. [168] introduced LogPPT to fine-tune RoBERTa for log parsing. In addition, an adaptive random sampling strategy was designed to select a small yet diverse training dataset. Ma et al. [136] compared in-context learning and fine-tuning using open-source LLMs such as Flan-T5 [169] and LLaMA [116] on log parsing. Zhi et al. [106] introduced YALP, which leverages the capabilities of ChatGPT (gpt-3.5-turbo) in conjunction with traditional methods — Longest Common Subsequence,without incorporating user labeling (zero-shot learning). Zhong et al. [170] proposed LogParser-LLM, which essentially blends a prefix tree and an LLM-based template extractor. This extractor parses log messages with different LLMs, including GPT-3.5-turbo, GPT-4, and Llama-2-13B, in either ICL or fine-tuning manner; the highest results were obtained using GPT-4 with ICL. Xiao et al. [104] introduced LogBatcher, which is a cost-effective LLM-based log parser based on GPT-3.5-Turbo. Similar to YALP, they control the cost of using closed-

source LLM by storing inferred messages in a basic cache. Moreover, it does not require any training by prompting the LLM with a group of high-diversity log messages to ensure that the LLM understands the diversity of the dataset in a zero-shot manner. Overall, the results of the above works align with our findings: the vast pretrained knowledge of LLMs enables data efficiency and robustness on unseen log templates.

3.7 Conclusion and Future Work

This chapter proposed a novel approach, FlexLog, for anomaly detection on unstable logs (ULAD), exploiting the synergy between Large Language Models (LLMs) and Machine Learning (ML) models via ensemble learning. FlexLog incorporates four base models, one LLM (Mistral), and three ML models (KNN, DT, and SLFN), which are trained on limited stable logs, reducing the usage of labeled data significantly. To classify unstable logs, FlexLog first processes unstable logs into log sequences through log parsing and partitioning. Then, FlexLog employs Retrieval-Augmented Generation (RAG) to fetch relevant information (if available) for these sequences, constructing context-enriched prompts. The fine-tuned LLM processes these prompts, while ML models use the log sequences directly for prediction. Finally, FlexLog combines the predictions of all the base models using majority voting to produce the final classification.

Our extensive experiments on two real-world and two synthesized datasets show that FlexLog achieves state-of-the-art effectiveness on all datasets while reducing the usage of labeled data by 62.87 pp to 78.43 pp, respectively. Further experiments on ADFA-U with varying limited training data size demonstrate that FlexLog maintains robust effectiveness under varying levels of data scarcity, except the extreme data scarcity scenario ($\mathcal{D}_{\#} = 50$), where all methods exhibit poor performance due to insufficient labeled data. FlexLog outperforms the top baseline in terms of F1 by 13 pp when the training dataset contains only 500 samples. However, experiments assessing time efficiency show that FlexLog trades off some time efficiency for this effectiveness but still manages to keep the inference time below one second per log sequence. This suggests FlexLog is applicable for most systems, except those with stringent latency requirements, such as high-frequency trading systems. Furthermore, in terms of memory efficiency, FlexLog 's cache memory remains **below 4** MB for most datasets (up to 19.6 MB for ADFA-U) confirming its **memory efficiency**. Finally, we conducted ablation studies on individual components of FlexLog, as well as evaluating alternative LLM choices. We confirmed the significant contributions of the cache-based inference, RAG, and ensemble learning with Mistral as LLM and KNN, DT, and SLFN as ML models.

In the future, we plan to further enhance the effectiveness of FlexLog by exploring more powerful open-source LLMs as base models, such as DeepSeek R1 [171], along with more advanced prompting techniques such as agent-based prompting [172]. Additionally, we wish to investigate different ensemble learning techniques that dynamically decide the optimal ensemble composition for a given log sequence. To enhance the generalizability of our caching mechanism, we plan to incorporate similarity-based retrieval in the future, allowing the model to infer labels from similar seen log sequences instead of identical ones.

Furthermore, to address the dataset imbalance observed in certain datasets, we aim to explore techniques like Generative Adversarial Networks (GANs) for data augmentation, potentially improving the performance of ML-based models within the ensemble.

Chapter 4

Conclusion

In this chapter, we summarize the thesis's contributions and discuss potential directions for future work.

4.1 Summary of Contributions

This thesis advances the state of the art in log-based software dependability analysis through two key objectives.

TO_1 (Evaluation of DL-based Failure Predictors): To address the need for systematic guidance on applying DL to log-based failure prediction, we proposed a modular architecture that can be configured with different combinations of embedding strategies and DL encoders. We examined three embedding strategies (Logkey2vec, BERT, and a hybrid of FastText and TF-IDF) and four major DL encoders (LSTM, BiLSTM, CNN, and transformer-based). To systematically assess performance across different datasets, we synthesized 1080 datasets with controlled characteristics, including dataset size, failure percentage, log sequence length, and failure pattern type. Our evaluation indicates that CNN-based encoders yield the highest accuracy across embedding strategies. In particular, CNN combined with Logkey2vec achieves strong performance, especially when the dataset size exceeds 350 or the failure percentage surpasses 7.5%. These findings provide actionable, dataset-specific guidelines for practitioners aiming to deploy DL-based failure predictors.

TO_2 (Robust and Data-efficient Anomaly Detection): To address the challenge of anomaly detection on unstable logs (ULAD), we designed FlexLog, a hybrid method that combines simple ML models (KNN, DT, and SLFN) with a LLM (Mistral) using ensemble learning. The approach incorporates caching and RAG to enhance further efficiency and robustness in scenarios with limited labeled data. We evaluated FlexLog on two real-world and two synthetic ULAD datasets. Results show that it consistently outperforms state-of-the-art baselines by at least 1.2 pp points in F1 score while reducing labeled data requirements by 62.87 pp. On ADFA-U, FlexLog improves F1 score by up to 13 pp when trained with only 500 samples. While the LLM (Mistral) affects the inference

time the most, inference time remains under one second per log sequence, confirming our method's practicality. Ablation studies further validated the positive impact of caching, RAG, and the hybrid ensemble design.

By fulfilling TO_1 and TO_2, this thesis contributes both a systematic study for understanding DL-based failure prediction and providing practical guidelines, and a robust, data-efficient approach for anomaly detection in unstable logs.

4.2 Discussion: Research Implications and Broader Context

The findings of this thesis have several implications for both research and practice.

4.2.1 Practical Implications

Failure Prediction. The systematic evaluation clarifies the trade-offs among embedding strategies and DL architectures, offering practitioners guidelines for selecting models based on dataset conditions. In practice, CNN+Logkey2vec proves to be a reliable choice when data availability is sufficient, while BERT, as a semantic embedding strategy, consistently delivers strong performance across diverse settings.

Anomaly Detection. FlexLog demonstrates that hybrid ML–LLM architectures can reduce reliance on large labeled datasets while maintaining high effectiveness. The integration of caching and RAG makes it more practical for deploying efficient anomaly detectors in real-world environments, especially where logs are unstable and labeled data is scarce.

4.2.2 Generalizability and Limitations

Failure Prediction. Our systematic evaluation highlights several conditions under which deep learning models for log-based failure prediction generalize well, as well as factors that limit their applicability:

- **Dataset characteristics.** Model effectiveness is strongly influenced by dataset size and failure percentage. CNN-based predictors with Logkey2vec embeddings consistently achieve high accuracy when the dataset size exceeds 350 instances or the failure percentage surpasses 7.5%. This provides clear guidance for practitioners on when DL-based predictors are most beneficial. Conversely, for small or highly imbalanced datasets, accuracy degrades substantially, limiting practical applicability.
- **Log sequence length.** Sequence length affects prediction only for specific encoder–embedding combinations. While CNNs remain robust across different lengths, RNN- and transformer-based models are more sensitive, suggesting their generalizability may be limited in domains with highly variable sequence lengths.

- **Noise and instability.** Our study primarily used synthetic datasets with controlled variations. Although these enable systematic evaluation, real-world logs may include mislabeled entries, parsing errors, or evolving templates, which could reduce prediction accuracy. Additional evaluation on diverse real-world systems is needed to confirm robustness under such conditions.
- **Embedding–encoder interactions.** Effectiveness depends not only on the encoder type but also on the embedding strategy. While BERT embeddings are strong overall, their advantages diminish when paired with CNNs, where Logkey2vec consistently performs best. This interaction must be considered when transferring models across domains.
- **Synthetic vs. real datasets.** Although results on real-world datasets were consistent with trends observed on synthetic data, synthetic generation cannot capture all characteristics of production systems. This may limit the external validity of findings until further real-world validation is conducted.

Anomaly Detection. While FlexLog demonstrates robustness and state-of-the-art effectiveness, several limitations affect its generalizability.

- **Token limits and sequence length.** Although our evaluation showed that token consumption remains below the input limits of current LLMs, extremely long log sequences may still pose challenges in some domains. Advances in LLM architectures with extended token windows can likely mitigate this limitation, but careful consideration of log partitioning strategies remains necessary in practice.
- **Data leakage risks.** As with any LLM-based approach, there is a residual risk that logs used for evaluation may overlap with the LLM's pretraining corpus. While we minimized this risk by including datasets released after Mistral's cut-off date (e.g., LOGEVOL-U, SYNEVOL-U), complete elimination cannot be guaranteed.
- **Dataset representativeness.** Although FlexLog was evaluated on both real-world and synthetic datasets, further validation across more diverse systems is needed. While our synthesized datasets mimic software and environment changes, they may not fully capture the diversity of real-world instability scenarios.
- **Extreme instability.** FlexLog remains robust and effective up to high levels of instability (e.g., 30% injection ratio of template or sequence level changes on SYNEVOL-U). Nevertheless, further evaluation on a broader range of real-world systems is necessary to fully assess the limitations of anomaly detectors when log instability is extreme.

4.3 Future Work

Building on TO_1 and TO_2, we identify the following research directions:

4.3.1 Failure Prediction (TO_1)

1. **Real-world evaluation.** Extend the evaluation of the best-performing DL+embedding configurations to large-scale, real-world log datasets.
2. **Impact of noise.** Assess the effect of mislabeled logs, evolving log formats, and log parsing errors on prediction accuracy.
3. **Timeliness metrics.** Complement F1 score with timeliness-oriented metrics such as lead time (number of log messages before failure).
4. **LLMs for Failure Prediction.** Explore the use of LLMs for failure prediction when DL models show limited effectiveness or when labeled data is scarce. Although LLMs require more computational resources and have lower time efficiency, they can still be beneficial when these trade-offs are acceptable in practice.

4.3.2 Anomaly Detection (TO_2)

1. **Integration of stronger LLMs.** Explore open-source LLMs such as DeepSeek R1 and future models with larger context windows.
2. **Adaptive ensembles.** Move beyond majority voting to dynamic ensemble strategies that tailor model composition per input.
3. **Similarity-based caching.** Extend caching to approximate matches for greater applicability.
4. **Data augmentation.** Apply GANs or synthetic log generation to mitigate data imbalance.
5. **Agentic anomaly detection.** Investigate iterative, repair–execute–feedback loops with LLMs for adaptive anomaly detection.
6. **Expanded evaluation of RAG.** Conduct further studies on RAG effectiveness as more datasets with contextual metadata become available to evaluate its generalizability across diverse domains.
7. **Qualitative user study.** Conduct a user study with system engineers and operators to assess the interpretability, usability, and practical integration of FlexLog in real monitoring workflows. Such qualitative insights can guide future improvements in model transparency and trustworthiness of automated anomaly detection methods.

4.4 Closing Remarks

This thesis achieves its two main objectives: (1) systematically investigating DL-based failure prediction approaches and providing dataset-specific practical guidelines, and (2)

designing a robust, data-efficient hybrid method for anomaly detection on unstable logs. Together, these contributions advance the state of the art for accurate and practical ML- and LLM-based log analysis. By releasing all datasets, benchmarks, and tools, this work also supports reproducibility and enables future research in the area.

References

[1] D. R. A. of Canada, https://alliancecan.ca/, 2016, accessed: March 2, 2025.

[2] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[3] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: How far are we?" in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1356–1367. [Online]. Available: https://doi.org/10.1145/3510003.3510155

[4] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A Survey on Automated Log Analysis for Reliability Engineering," *ACM Computing Surveys*, vol. 54, no. 6, 2021.

[5] T. P. Carvalho, F. A. A. M. N. Soares, R. Vita, R. da P. Francisco, J. P. Basto, and S. G. S. Alcalá, "A systematic literature review of machine learning methods applied to predictive maintenance," *Computers & Industrial Engineering*, vol. 137, p. 106024, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835219304838

[6] A. Das, F. Mueller, C. Siegel, and A. Vishnu, "Desh: Deep learning for system health prediction of lead times to failure in HPC," *HPDC 2018 - Proceedings of the 2018 International Symposium on High-Performance Parallel and Distributed Computing*, pp. 40–51, 2018.

[7] A. Das, F. Mueller, and B. Rountree, "Aarohi: Making Real-Time Node Failure Prediction Feasible," *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium, IPDPS 2020*, pp. 1092–1101, 2020.

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 117–132. [Online]. Available: https://doi.org/10.1145/1629575.1629587

[9] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 413–422.

[10] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8.

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 1285–1298, 2017.

[12] B. Yu, J. Yao, Q. Fu, Z. Zhong, H. Xie, Y. Wu, Y. Ma, and P. He, "Deep learning or classical machine learning? an empirical study on log-based anomaly detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623308

[13] Y. Huo, C. Lee, Y. Su, S. Shan, J. Liu, and M. R. Lyu, "Evlog: Identifying anomalous logs over software evolution," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 391–402. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSRE59848.2023.00018

[14] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[15] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 426–435, 2003.

[16] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI*, vol. 19, no. 7, 2019, pp. 4739–4745.

[17] F. Hadadi, J. H. Dawes, D. Shin *et al.*, "Systematic evaluation of deep learning models for log-based failure prediction," *Empirical Software Engineering*, vol. 29, p. 105, 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10501-4

[18] F. Hadadi, Q. Xu, D. Bianculli, and L. Briand, "Llm meets ml: Data-efficient anomaly detection on unstable logs," 2025. [Online]. Available: https://arxiv.org/abs/2406.07467

[19] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[20] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[21] Y. Chen, X. Yang, Q. Lin, D. Zhang, H. Dong, Y. Xu, H. Li, Y. Kang, H. Zhang, F. Gao, Z. Xu, and Y. Dang, "Outage prediction and diagnosis for cloud service systems," *The Web Conference 2019 - Proceedings of the World Wide Web Conference, WWW 2019*, pp. 2659–2665, 2019.

[22] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," *IEEE Access*, vol. 6, pp. 21 929–21 940, 2018.

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, nov 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[25] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[26] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1511.08458

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[28] D. Shin, D. Bianculli, and L. Briand, "Prins: Scalable model inference for component-based system logs," *Empirical Softw. Engg.*, vol. 27, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10111-4

[29] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 301–310.

[30] J. Bogatinovski, S. Nedelkoski, L. Wu, J. Cardoso, and O. Kao, "Failure identification from unstable log data using deep learning," *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 346–355, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247996709

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Comput.*, vol. 12, no. 10, p. 2451–2471, oct 2000. [Online]. Available: https://doi.org/10.1162/089976600300015015

[33] W. Meng, Y. Liu, Y. Huang, S. Zhang, F. Zaiter, B. Chen, and D. Pei, "A semantic-aware representation framework for online log analysis," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–7.

[34] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," 2015. [Online]. Available: https://arxiv.org/abs/1508.01991

[35] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. [Online]. Available: https://aclanthology.org/D14-1181

[36] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[37] V. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2021, pp. 492–504. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE51524.2021.9678773

[38] S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan, "Hitanomaly: Hierarchical transformers for anomaly detection in system log," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2064–2076, 2020.

[39] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," *Proceedings - IEEE International Conference on Data Mining, ICDM*, vol. 2020-Novem, no. Icdm, pp. 1196–1201, 2020.

[40] V. P. Dwivedi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson, "Graph neural networks with learnable structural and positional representations," *CoRR*, vol. abs/2110.07875, 2021. [Online]. Available: https://arxiv.org/abs/2110.07875

[41] X. Wu, H. Li, and F. Khomh, "On the effectiveness of log representation for log-based anomaly detection," 2023.

[42] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[43] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 2016, pp. 207–218. [Online]. Available: https://doi.org/10.1109/ISSRE.2016.21

[44] A. Rajaraman, J. Leskovec, and J. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 01 2014.

[45] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations*, 2013.

[46] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *Neural Information Processing Systems*, 2019.

[47] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[48] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[49] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," 2016.

[50] C. C. Foundation, "Common crawl corpus," 2023. [Online]. Available: https://commoncrawl.org/

[51] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," pp. 1448–1460, 2021.

[52] Y. Xie, H. Zhang, and M. A. Babar, "Loggd: Detecting anomalies from system logs with graph neural networks," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 299–310.

[53] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang, "Predicting node failure in cloud service systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 480–490. [Online]. Available: https://doi.org/10.1145/3236024.3236060

[54] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[55] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 499–508. [Online]. Available: https://doi.org/10.1145/1081870.1081927

[56] B. Russo, G. Succi, and W. Pedrycz, "Mining system logs to learn error predictors: a case study of a telemetry system," *Empirical Software Engineering*, vol. 20, no. 4, pp. 879–927, 2015.

[57] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, Y. Chen, H. Dong, X. Qu, and L. Song, "Prefix: Switch failure prediction in datacenter networks," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 1, pp. 2:1–2:29, 2018. [Online]. Available: https://doi.org/10.1145/3179405

[58] X. Liu, Y. He, H. Liu, J. Zhang, B. Liu, X. Peng, J. Xu, J. Zhang, A. Zhou, P. Sun, K. Zhu, A. Nishi, D. Zhu, and K. Zhang, *Smart Server Crash Prediction in Cloud Service Data Center*. 2020 19th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2020.

[59] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2020-Octob, pp. 92–103, 2020.

[60] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and generating test cases using observer automata," *Lecture Notes in Computer Science*, vol. 3395, pp. 125–139, 2005.

[61] A. Bombarda and A. Gargantini, "An Automata-Based Generation Method for Combinatorial Sequence Testing of Finite State Machines," *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, pp. 157–166, 2020.

[62] F. Kluge, C. Rochange, and T. Ungerer, "EMSBench: Benchmark and Testbed for Reactive Real-Time Systems," *Leibniz Transactions on Embedded Systems*, vol. 4, no. 2, pp. 02–1–02:23, 2017. [Online]. Available: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v004-i002-a002

[63] S. Krstić and J. Schneider, *A Benchmark Generator for Online First-Order Monitoring*. Springer International Publishing, 2020, vol. 12399 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-60508-7_27

[64] D. Basin, T. Dardinier, L. Heimes, S. Krstić, M. Raszyk, J. Schneider, and D. Traytel, "A formally verified, optimized monitor for metric first-order dynamic logic," in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part*

I. Berlin, Heidelberg: Springer-Verlag, 2020, p. 432–453. [Online]. Available: https://doi.org/10.1007/978-3-030-51074-9_25

[65] D. Weijie, L. Yunyi, Z. Jing, and S. Xuchen, "Long text classification based on bert," in *2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*, vol. 5, 2021, pp. 1147–1151.

[66] M. Ding, C. Zhou, H. Yang, and J. Tang, "Cogltx: Applying bert to long texts," in *Neural Information Processing Systems*, 2020.

[67] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" in *Chinese Computational Linguistics*, M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, Eds. Cham: Springer International Publishing, 2019, pp. 194–206.

[68] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research*, vol. 15, pp. 3133–3181, 2014.

[69] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, 1984.

[70] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0192-5

[71] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? An empirical analysis of software failures in the OpenStack cloud computing platform," *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 200–211, 2019.

[72] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the Trade*. Springer, 1998, pp. 55–69.

[73] G. Upton and I. Cook, *A Dictionary of Statistics*, ser. Oxford Paperback Reference. OUP Oxford, 2008. [Online]. Available: https://books.google.ca/books?id=u97pzxRjaCQC

[74] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[75] Y. Chen, L. Li, W. Li, Q. Guo, Z. Du, and Z. Xu, *AI Computing Systems: An Application Driven Perspective*. Elsevier Science, 2022. [Online]. Available: https://books.google.ca/books?id=RSWJEAAAQBAJ

[76] A. Tauber, "exrex: Irregular methods for regular expressions." 2018, accessed 2025-08-24. [Online]. Available: https://github.com/asciimoo/exrex

[77] R. P. Package, 2019, accessed 2025-08-24. [Online]. Available: https://docs.python.org/3/library/random.html

[78] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[79] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A Large Collection of System Log Datasets Towards Automated Log Analytics," *CoRR*, vol. abs/2008.06448, 2020. [Online]. Available: https://arxiv.org/abs/2008.06448

[80] P. E. Black, "Strongly connected component," *Dictionary of Algorithms and Data Structures*, 2020. [Online]. Available: https://www.nist.gov/dads/HTML/stronglyConnectedCompo.html

[81] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," IEEE, pp. 33–40, 2017.

[82] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. Piscataway, NJ, USA: IEEE Press, 2018, pp. 167–16 710.

[83] P. Xu, D. Kumar, W. Yang, W. Zi, K. Tang, C. Huang, J. C. K. Cheung, S. Prince, and Y. Cao, "Optimizing deeper transformers on small datasets," in *Annual Meeting of the Association for Computational Linguistics*, 2020.

[84] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *ArXiv*, vol. abs/1506.00019, 2015.

[85] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320317304120

[86] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, pp. 1263–1284, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:206742563

[87] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[88] F. Hadadi, J. Dawes, D. Shin, D. Bianculli, and L. Briand, "Replication Package," 5 2024. [Online]. Available: https://figshare.com/articles/software/Replication_Package/22219111

[89] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of aiops methods for failure management," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 6, nov 2021. [Online]. Available: https://doi.org/10.1145/3483424

[90] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys*, vol. 42, no. 3, 2010.

[91] H. Krasner, "The cost of poor software quality in the us: A 2020 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pp. 1–46, 2021.

[92] M. Hejazi and Y. P. Singh, "One-class support vector machines approach to anomaly detection," *Applied Artificial Intelligence*, vol. 27, no. 5, pp. 351–366, 2013.

[93] X. Wang, L. Yang, D. Li, L. Ma, Y. He, J. Xiao, J. Liu, and Y. Yang, "Maddc: Multi-scale anomaly detection, diagnosis and correction for discrete event logs," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 769–784. [Online]. Available: https://doi.org/10.1145/3564625.3567972

[94] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log Clustering Based Problem Identification for Online Service Systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE - Companion Volume*. Austin, TX, USA: ACM, 2016, pp. 102–111. [Online]. Available: https://doi.org/10.1145/2889160.2889232

[95] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, pp. 37–44, 2010.

[96] B. Yu, J. Yao, Q. Fu, Z. Zhong, H. Xie, Y. Wu, Y. Ma, and P. He, "Deep learning or classical machine learning? an empirical study on log-based anomaly detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623308

[97] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 588–597, 12 2009.

[98] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN2007*. IEEE, 2007, pp. 575–584.

[99] M. He, T. Jia, C. Duan, H. Cai, Y. Li, , and G. Huang, "Llmelog: An approach for anomaly detection based on llm-enriched log events," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. Tsukuba, Japan: IEEE Computer Society, 2024.

[100] Y. Liu, S. Tao, W. Meng, F. Yao, X. Zhao, and H. Yang, "Logprompt: Prompt engineering towards zero-shot and interpretable log analysis," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 364–365. [Online]. Available: https://doi.org/10.1145/3639478.3643108

[101] W. Zhang, T. Jia, C. Duan, H. Cai, Y. Li, , and G. Huang, "Leveraging rag-enhanced large language model for semi-supervised log anomaly detection," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. Tsukuba, Japan: IEEE Computer Society, 2024.

[102] M. Mosbach, T. Pimentel, S. Ravfogel, D. Klakow, and Y. Elazar, "Few-shot fine-tuning vs. in-context learning: A fair comparison and evaluation," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 12 284–12 314. [Online]. Available: https://aclanthology.org/2023.findings-acl.779

[103] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "Lilac: Log parsing using llms with adaptive parsing cache," 2024.

[104] Y. Xiao, V.-H. Le, and H. Zhang, "Demonstration-free: Towards more practical log parsing with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 153–165. [Online]. Available: https://doi.org/10.1145/3691620.3694994

[105] V. Le and H. Zhang, "Log parsing: How far can chatgpt go?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 1699–1704. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00206

[106] C. Zhi, L. Cheng, M. Liu, X. Zhao, Y. Xu, and S. Deng, "Llm-powered zero-shot online log parsing," in *2024 IEEE International Conference on Web Services (ICWS)*, 2024, pp. 877–887.

[107] C. Irugalbandara, A. Mahendra, R. Daynauth, T. K. Arachchige, J. Dantanarayana, K. Flautner, L. Tang, Y. Kang, and J. Mars, "Scaling down to scale up: A cost-benefit analysis of replacing openai's llm with open source slms in production," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 280–291.

[108] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021. [Online]. Available: https://doi.org/10.1145/3460345

[109] S. Kabinna, W. Shang, C.-P. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 326–337.

[110] V.-H. Le and H. Zhang, "Log-based anomaly detection with deep learning: how far are we?" in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1356–1367. [Online]. Available: https://doi.org/10.1145/3510003.3510155

[111] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[112] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu *et al.*, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[113] OpenAI, "Fine-tuning: preparing your dataset," 2024, accessed 2024-01-02. [Online]. Available: https://platform.openai.com/docs/guides/fine-tuning

[114] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. E. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. J. Lowe, "Training language models to follow instructions with human feedback," *ArXiv*, vol. abs/2203.02155, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246426909

[115] L. O. J. W. X. J. D. A. C. L. W. P. M. P. C. J. L. R. Lowe *et al.*, "Gpt-4 technical report," 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257532815

[116] H. Touvron, L. Martin, K. Stone, and P. Albert, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[117] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[118] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-efficient fine-tuning for large models: A comprehensive survey," 2024.

[119] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[120] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023. [Online]. Available: https://arxiv.org/abs/2305.14314

[121] M. Winteringham, *Software Testing with Generative AI*. Manning Publications, 2024.

[122] R. Polikar, "Ensemble learning," *Ensemble machine learning: Methods and applications*, pp. 1–34, 2012.

[123] Z.-H. Zhou and Z.-H. Zhou, *Ensemble learning*. Springer, 2021.

[124] Y. Zhang, J. Liu, and W. Shen, "A review of ensemble learning algorithms used in remote sensing applications," *Applied Sciences*, vol. 12, no. 17, p. 8654, 2022.

[125] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, 2013, pp. 4487–4492.

[126] Intel, "Hibench," https://github.com/Intel-bigdata/HiBench, 2021.

[127] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering." *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[128] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds. ACL, 2014, pp. 1724–1734. [Online]. Available: https://doi.org/10.3115/v1/d14-1179

[129] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," 2016.

[130] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[131] E. Fix and J. L. Hodges, "Discriminatory analysis - nonparametric discrimination: Consistency properties," *International Statistical Review*, vol. 57, p. 238, 1989. [Online]. Available: https://api.semanticscholar.org/CorpusID:120323383

[132] M. Y. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 36–43, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:56849250

[133] G. Huang, Y. Q. Chen, and H. A. Babri, "Classification ability of single hidden layer feedforward neural networks," *IEEE transactions on neural networks*, vol. 11 3, pp. 799–801, 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:1852628

[134] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[135] G. Chu, J. Wang, Q. Qi, H. Sun, S. Tao, and J. Liao, "Prefix-graph: A versatile log parsing approach merging prefix tree with probabilistic graph," in *Proceedings of the 37th International Conference on Data Engineering (ICDE)*. Virtual Event: IEEE, 2021, pp. 2411–2422. [Online]. Available: https://ieeexplore.ieee.org/document/9458609

[136] Z. Ma, A. R. Chen, D. J. Kim, T.-H. Chen, and S. Wang, "Llmparser: An exploratory study on using large language models for log parsing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639150

[137] Z. A. Khan, D. Shin, D. Bianculli, and L. C. Briand, "Impact of log parsing on deep learning-based anomaly detection," *Empirical Software Engineering*, vol. 29, no. 6, Aug. 2024. [Online]. Available: http://dx.doi.org/10.1007/s10664-024-10533-w

[138] MistralAI, "Mistral technologies," 2024, accessed 2024-09-24. [Online]. Available: https://mistral.ai/technology/

[139] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[140] OpenAI, "Openai models," 2024, accessed 2024-05-10. [Online]. Available: https://platform.openai.com/docs/models

[141] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, and A. Letman, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[142] S. Yang, N. Kassner, E. Gribovskaya, S. Riedel, and M. Geva, "Do large language models perform latent multi-hop reasoning without exploiting shortcuts?" 2024. [Online]. Available: https://arxiv.org/abs/2411.16679

[143] Unsloth, "Unsloth fine-tuning package," 2024, accessed 2024-11-30. [Online]. Available: https://github.com/unslothai/unsloth

[144] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, R. Meersman, Z. Tari, and D. C. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 986–996.

[145] S. Zhang, "Challenges in knn classification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 10, pp. 4663–4675, 2021.

[146] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel, "A simple neural attentive meta-learner," 2018. [Online]. Available: https://arxiv.org/abs/1707.03141

[147] W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, and S. Yan, "Metaformer is actually what you need for vision," 2022. [Online]. Available: https://arxiv.org/abs/2111.11418

[148] F. Hadadi, Q. Xu, D. Bianculli, and L. Briand, "Replication Package," 10 2025. [Online]. Available: https://figshare.com/s/2a8a747a5b6029ce93f4

[149] E. Diagboya, *Infrastructure Monitoring with Amazon CloudWatch: Effectively monitor your AWS infrastructure to optimize resource allocation, detect anomalies, and set automated actions*. Packt Publishing Ltd, 2021.

[150] M. Collier and R. Shahan, *Microsoft azure essentials-fundamentals of azure*. Microsoft Press, 2015.

[151] J. Turnbull, *Monitoring with Prometheus*. Turnbull Press, 2018.

[152] H. O. Bello, A. B. Ige, and M. N. Ameyaw, "Deep learning in high-frequency trading: conceptual challenges and solutions for real-time fraud detection," *World Journal of Advanced Engineering Technology and Sciences*, vol. 12, no. 02, pp. 035–046, 2024.

[153] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," IEEE, pp. 1–10, 2010.

[154] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[155] H. Kaur, H. S. Pannu, and A. K. Malhi, "A systematic review on imbalanced data challenges in machine learning: Applications and solutions," *ACM computing surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.

[156] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: overview study and experimental results," in *2020 11th international conference on information and communication systems (ICICS)*. IEEE, 2020, pp. 243–248.

[157] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[158] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," pp. 4171–4186, 2019. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[159] Q. Team, "Qwen2.5-1m: Deploy your own qwen with context length up to 1m tokens," January 2025. [Online]. Available: https://qwenlm.github.io/blog/qwen2.5-1m/

[160] A. Yang, B. Yu, C. Li, D. Liu, F. Huang, H. Huang, J. Jiang, J. Tu, J. Zhang, J. Zhou, J. Lin, K. Dang, K. Yang, L. Yu, M. Li, M. Sun, Q. Zhu, R. Men, T. He, W. Xu, W. Yin, W. Yu, X. Qiu, X. Ren, X. Yang, Y. Li, Z. Xu, and Z. Zhang, "Qwen2.5-1m technical report," *arXiv preprint arXiv:2501.15383*, 2025.

[161] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[162] M. Landauer, S. Onder, F. Skopik, and M. Wurzenberger, "Deep learning for anomaly detection in log data: A survey," *Machine Learning with Applications*, vol. 12, p. 100470, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666827023000233

[163] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Divlog: Log parsing with prompt enhanced in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639155

[164] X. Han, S. Yuan, and M. Trabelsi, "Loggpt: Log anomaly detection via gpt," in *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 1117–1122.

[165] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," March 2023. [Online]. Available: https://lmsys.org/blog/2023-03-30-vicuna/

[166] L. Chen, G. Zhang, and H. Zhou, "Fast greedy map inference for determinantal point process to improve recommendation diversity," 2018. [Online]. Available: https://arxiv.org/abs/1709.05135

[167] C. Pei, Z.Liu, J. Li, E. Zhang, L. Zhang, H. Zhang, W. Chen, D. Pei, and G. Xie, "Self-evolutionary group-wise log parsing based on large language model," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. Tsukuba, Japan: IEEE Computer Society, 2024.

[168] V.-H. Le and H. Zhang, "Log parsing with prompt-based few-shot learning," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 2438–2449. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00204

[169] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," 2022.

[170] A. Zhong, D. Mo, G. Liu, J. Liu, Q. Lu, Q. Zhou, J. Wu, Q. Li, and Q. Wen, "Logparser-llm: Advancing efficient log parsing with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2408.13727

[171] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, and X. Bi, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[172] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[173] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

Appendix

A.1 Alternative Ensembling Strategies

FlexLog employs a majority voting strategy, where in the case of a tie between base models, the label is set to *normal*, as anomalies are rare. This ensemble approach uses binary labels for voting rather than anomaly confidence scores, allowing us to adopt base models that directly output the final label, such as our fine-tuned version of Mistral. Alternative ensembling strategies include the alternative majority voting method, which assigns a random label in the case of a tie, and state-of-the-art meta-learning approaches that adaptively learn from base model performance on validation data. Specifically, SNAIL [146] is an attentive CNN-based meta-learner, while MetaFormer [147] introduces a novel attention module called *token mixer*; both have shown significant improvements over traditional meta-learning approaches. We experimented with these methods using their original implementation code.

Table A.1 reports the F1 scores of FlexLog using different ensemble strategies, including the original majority voting (*Majority Voting (alternative)*), and two meta-learning approaches, *SNAIL* and *MetaFormer*. Experimental results show that FlexLog's majority voting consistently outperforms these alternatives with an average difference of 5 pp, 3 pp, 2 pp, and 3 pp, respectively, for ADFA-U, LOGEVOL-U, SynHDFS-U, and LOGEVOL-U. Mann-Whitney U tests at the dataset level further confirm that the differences in F1 score between FlexLog and the alternative strategies are statistically significant across all four datasets, except for MetaFormer on LOGEVOL-U. Overall, the modified majority voting in FlexLog remains the recommended strategy due to its simplicity and effectiveness.

Table A.1: F1 scores of using alternative Ensembling Strategies in FlexLog.

Ensembling Config	**ADFA-U**							**LOGEVOL-U**			**SynHDFS-U**	**SYNEVOL-U**
	adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
Majority Voting (FlexLog)	**0.718**	**0.784**	**0.723**	**0.642**	**0.682**	**0.672**	**0.704**	**0.982**	**0.892**	**0.937**	**0.972**	**0.971**
Majority Voting (alternative)	0.641	0.711	0.691	0.615	0.656	0.635	0.650*	0.964	0.840	0.902*	0.936*	0.929*
SNAIL	0.671	0.714	0.683	0.624	0.628	0.651	0.661*	0.965	0.854	0.909*	0.958*	0.949*
MetaFormer	0.666	0.706	0.691	0.615	0.607	0.663	0.658*	0.976	0.866	0.921	0.954*	0.951*

* FlexLog yields a significant higher F1 score compared to using the alternative ensembling strategy.

A.2 Impact of Deduplication

Table A.2 reports the F1 scores of FlexLog and baseline models evaluated on both deduplicated and original test data. The column "dedup" indicates whether the test set was deduplicated from the training set. Overall, all models exhibit inflated F1 scores when the test set is not deduplicated from the training set, highlighting the impact of data leakage. For instance, NeuralLog shows an average inflation of 16 pp, 2 pp, 1 pp, and 24 pp on ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U, respectively. Mann-Whitney U tests confirm that the differences are statistically significant for Neurallog. In contrast to the most effective baseline (LightAD), FlexLog keeps the inflation in F1 score below 2 pp across all datasets, where the difference between FlexLog's performance on deduplicated and original test data is statistically insignificant, confirmed by Mann-Whitney U tests.

Table A.2: Effectiveness of FlexLog and baselines for ULAD on ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U with and without deduplication.

Model	Dedup	ADFA-U							LOGEVOL-U			SynHDFS-U	SYNEVOL-U
		adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
FlexLog	Yes	0.718	0.784	**0.723**	0.642	0.682	0.672	0.704	0.982	0.892	**0.937**	0.972	**0.971**
	No	0.723	0.790	0.726	0.652	0.701	0.673	0.711	0.996	0.895	0.945	0.974	0.987
LightAD	Yes	0.725	0.754	0.666	0.639	0.679	0.602	0.677	0.980	0.829	0.898	0.959	0.956
	No	0.745	0.778	0.745	0.646	0.695	0.618	0.704*	0.995	0.876	0.935*	0.968	0.981*
NeuralLog	Yes	0.412	0.388	0.368	0.511	0.461	0.457	0.433	0.948	0.834	0.891	0.946	0.765
	No	0.606	0.681	0.601	0.570	0.645	0.492	0.599*	0.961	0.859	0.910*	0.951	0.905*
LogRobust	Yes	0.524	0.408	0.374	0.558	0.651	0.449	0.494	0.927	0.757	0.833	0.760	0.941
	No	0.636	0.597	0.504	0.662	0.660	0.496	0.592*	0.981	0.789	0.885*	0.929*	0.966*
CNN	Yes	0.641	0.750	0.750	0.635	0.711	0.621	0.685	0.980	0.840	0.910	0.942	0.918
	No	0.703	0.765	0.777	0.644	0.724	0.634	0.707*	0.989	0.863	0.925	0.961*	0.925
PLELog	Yes	0.361	0.388	0.473	0.430	0.253	0.233	0.356	0.709	0.165	0.437	0.499	0.164
	No	0.405	0.428	0.494	0.443	0.311	0.277	0.393*	0.761	0.223	0.492*	0.528*	0.236*
LogAnomaly	Yes	0.291	0.451	0.480	0.422	0.218	0.343	0.368	0.310	0.216	0.263	0.446	0.304
	No	0.305	0.464	0.486	0.399	0.237	0.351	0.373	0.619	0.212	0.415*	0.498*	0.336*
DeepLog	Yes	0.292	0.481	0.458	0.339	0.253	0.353	0.363	0.367	0.122	0.244	0.741	0.253
	No	0.340	0.499	0.450	0.341	0.249	0.369	0.374	0.685	0.141	0.413*	0.776*	0.291*
LogCluster	Yes	0.334	0.418	0.461	0.348	0.175	0.304	0.340	0.430	0.485	0.458	0.519	0.714
	No	0.326	0.431	0.523	0.317	0.211	0.336	0.357*	0.798	0.614	0.706*	0.759*	0.786*
PCA	Yes	0.165	0.144	0.140	0.158	0.241	0.197	0.174	0.360	0.108	0.234	0.404	0.103
	No	0.155	0.152	0.163	0.277	0.211	0.198	0.192*	0.454	0.097	0.275*	0.567*	0.189*

* The same model shows a significant F1 score difference between deduplicated and original test data.

These results indicate that FlexLog delivers more reliable and robust performance. To avoid any risk of inflated results, all reported performances in this paper are based on deduplicated test data.

A.3 Baselines with Limited Data

Table A.3 presents the effectiveness of FlexLog compared to baselines when all models are trained on the same limited dataset. FlexLog consistently achieves the highest F1 score across all datasets under this setting. For instance, FlexLog outperforms the supervised baselines LightAD, NeuralLog, LogRobust, and CNN by 10 pp, 16 pp, 30 pp, and 14 pp, respectively, on average for ADFA-U. This advantage is important because supervised models such as NeuralLog and CNN depend on large labeled datasets to achieve high accuracy. Mann-Whitney U tests confirm that the observed performance gaps between FlexLog and each baseline are statistically significant across all datasets, demonstrating that none of the baselines match FlexLog 's performance with limited data. Notably, FlexLog trained with limited data even outperforms baselines trained with full datasets in terms of predictive effectiveness. Detailed results are presented and discussed in RQ1 (§ 3.5.1).

A.4 Effectiveness on SYNEVOL-U with Sequence Level Changes

Table A.4 presents the effectiveness of FlexLog compared to baselines on SYNEVOL-U under varying log sequence instability levels. FlexLog consistently achieves the highest

Table A.3: Effectiveness of FlexLog and Baselines Trained with Limited Data on ADFA-U, LOGEVOL-U, SynHDFS-U, and SYNEVOL-U.

Model	ADFA-U							LOGEVOL-U			SynHDFS-U	SYNEVOL-U
	adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
FlexLog	**0.718**	**0.784**	**0.723**	**0.642**	**0.682**	**0.672**	**0.704**	**0.982**	**0.892**	**0.937**	**0.972**	**0.971**
LightAD	0.587	0.720	0.633	0.564	0.520	0.557	0.596*	0.973	0.821	0.897*	0.925*	0.915*
NeuralLog	0.607	0.645	0.630	0.577	0.438	0.328	0.537*	0.975	0.232	0.603*	0.914*	0.270*
LogRobust	0.393	0.551	0.424	0.392	0.299	0.374	0.405*	0.851	0.254	0.552*	0.930*	0.903*
CNN	0.534	0.703	0.603	0.545	0.422	0.549	0.559*	0.819	0.217	0.518*	0.925*	0.863*
PLELog	0.495	0.457	0.163	0.144	0.102	0.322	0.280*	0.309	0.313	0.311*	0.378*	0.160*
LogAnomaly	0.310	0.359	0.346	0.345	0.178	0.274	0.302*	0.298	0.105	0.201*	0.152*	0.251*
DeepLog	0.261	0.358	0.409	0.287	0.154	0.272	0.290*	0.355	0.061	0.208*	0.188*	0.219*
LogCluster	0.226	0.340	0.363	0.281	0.185	0.254	0.274*	0.460	0.321	0.390*	0.547*	0.445*
PCA	0.077	0.136	0.187	0.164	0.037	0.066	0.111*	0.134	0.051	0.092*	0.103*	0.120*

* FlexLog yields a significant higher F1 score than compared baselines.

F1 score across all injection ratios, surpassing LightAD by 1.3 pp on average (98.2% – 96.9%). A Mann-Whitney U test confirms this difference is statistically insignificant, indicating comparable effectiveness between FlexLog and LightAD. However, FlexLog achieves this performance while being trained on only 22.96 % of unique log sequences, reducing usage of labeled data by 77.04 pp. Unlike in SynHDFS-U, where only LightAD and FlexLog remain robust across instability levels, all supervised methods—including FlexLog, LightAD, NeuralLog, LogRobust, and CNN—do not show a strong correlation between performance and increasing log instability in SYNEVOL-U. In contrast, semi-supervised and unsupervised methods exhibit a clear decline in precision, recall, and F1 score as instability increases. One possible reason is that changes at the log sequence level in SYNEVOL-U often involve minor modifications, such as adding or removing a single template, which may have a limited impact on ULAD. For instance, at a 30 % injection ratio, 55 % of changes involve just one template, making the overall sequence structure relatively stable despite modifications.

A.5 Effectiveness of Few-shot ICL for FlexLog

Table A.5 presents the effectiveness of FlexLog, which leverages Parameter-Efficient Fine-Tuning (PEFT), compared to FlexLog where LLM (Mistral) leverages a few-shot In-Context Learning (ICL) with intact LLM weights, and a few examples from the training data provided in the input prompt. Column "Learning Method" represents the learning strategy for the LLM, either PEFT or ICL. For few-shot ICL, we followed the same prompt design and inserted the examples (shots) in the relevant part of the prompt. In this way, the prompt includes k examples and their labels, denoted by $example_i$ and $label_i$ for $i \in \{1, \ldots, k\}$, respectively. We set k to 4, 50, 4, 100, and 6 for ADFA-U, LOGEVOL-U Hadoop, LOGEVOL-U Spark, SynHDFS-U, and SYNEVOL-U, respectively, given the models' maximum input token limit (128 000 for Mistral) and the maximum token consumption of each dataset, discussed in § 3.5.5.1.

For each dataset, instead of random selection, we selected samples from its training set by applying the few-shot sampling algorithm based on Mean Shift Clustering [173].

Table A.4: Effectiveness of FlexLog and baselines under different sequence-level injection ratios on SYNEVOL-U.

Data	Unstable	M	limited data	full training set								
				Supervised				Semi-S	Unsupervised			
			FlexLog	LightAD	NeuralLog	LogRobust	CNN	PLELog	LogAnomaly	DeepLog	LogCluster	PCA
0%	No	P	0.999	0.999	0.999	0.941	0.999	0.172	0.501	0.512	0.771	0.072
		R	0.969	0.939	0.636	0.969	0.878	0.129	0.393	0.443	0.818	0.471
		F1	**0.984**	0.968	0.777	0.952	0.935	0.243	0.441	0.475	0.794	0.125
5%	Yes	P	0.999	0.999	0.999	0.969	0.999	0.179	0.388	0.384	0.783	0.057
		R	0.971	0.942	0.628	0.914	0.885	0.141	0.440	0.428	0.828	0.457
		F1	**0.985**	0.970	0.771	0.941	0.939	0.158	0.394	0.405	0.805	0.102
10%	Yes	P	0.999	0.999	0.999	0.971	0.999	0.177	0.326	0.271	0.769	0.072
		R	0.973	0.945	0.648	0.918	0.891	0.145	0.459	0.432	0.810	0.471
		F1	**0.986**	0.972	0.786	0.944	0.942	0.160	0.382	0.333	0.789	0.125
15%	Yes	P	0.999	0.999	0.999	0.971	0.999	0.163	0.224	0.229	0.769	0.063
		R	0.973	0.945	0.621	0.918	0.891	0.141	0.351	0.378	0.810	0.475
		F1	**0.986**	0.972	0.766	0.944	0.942	0.151	0.273	0.285	0.789	0.112
20%	Yes	P	0.999	0.999	0.999	0.973	0.999	0.189	0.205	0.200	0.804	0.053
		R	0.975	0.951	0.583	0.902	0.878	0.154	0.365	0.439	0.804	0.470
		F1	**0.987**	0.975	0.738	0.936	0.935	0.170	0.263	0.274	0.804	0.095
25%	Yes	P	0.999	0.999	0.999	0.975	0.999	0.167	0.180	0.165	0.800	0.071
		R	0.953	0.930	0.651	0.930	0.906	0.141	0.441	0.418	0.837	0.511
		F1	**0.976**	0.963	0.788	0.952	0.951	0.153	0.256	0.236	0.818	0.125
30%	Yes	P	0.999	0.999	0.999	0.972	0.972	0.174	0.162	0.125	0.659	0.058
		R	0.950	0.925	0.600	0.900	0.875	0.145	0.475	0.425	0.775	0.434
		F1	**0.974**	0.961	0.750	0.935	0.921	0.158	0.242	0.193	0.712	0.102
Average	Yes	P	0.999	0.999	0.999	0.972	0.994	0.175	0.247	0.229	0.764	0.062
		R	0.966	0.94	0.622	0.914	0.888	0.144	0.422	0.42	0.811	0.47
		F1	**0.982**	0.969	0.767*	0.942*	0.938*	0.158*	0.302*	0.288*	0.786*	0.110 *

* FlexLog yields a significant higher F1-score than compared baseline.

Following Ma et al. [136], we first apply Mean Shift to the processed logs to obtain clusters. We choose Mean Shift because, unlike k-means, it does not require specifying the number of clusters in advance; however, other clustering algorithms could also be used. Next, we sort clusters in descending order of size. We then sample one log from each cluster and iterate through the clusters (from largest to smallest). We repeat this sampling until reaching the desired number of samples. By sampling iteratively from the largest clusters, the procedure balances diversity (representation across clusters) and coverage (capturing most frequent patterns, represented by larger clusters). These few-shot prompts were then evaluated across all datasets for ULAD. It is important to note that, for fairness, all other base models in FlexLog remain unchanged and are trained on the same limited training data across both settings.

The results in Table A.5 show that FlexLog with a fine-tuned LLM (denoted as *PEFT*) consistently outperforms FlexLog with few-shot ICL in terms of F1 score, with a substantial performance gap ranging from 5.7 pp to 19.9 pp. Mann–Whitney U tests confirm that these differences are statistically significant. Overall, PEFT enables the LLM to leverage more labeled data, and once fine-tuned, the model can efficiently label test data without repeatedly requiring few-shot examples, which otherwise reduces time efficiency. In the future, the few-shot ICL strategy could be improved by developing algorithms that dynamically tailor the selected examples for each test log sequence and by breaking them into multiple prompts, allowing more examples to be provided, especially when logs are

long.

Table A.5: F1 scores of using Few-shot ICL compared to PEFT) for LLM in FlexLog.

Learning Method	ADFA-U							LOGEVOL-U			SynHDFS-U	SYNEVOL-U
	adduser	hydraFTP	hydraSSH	java	meter	web	average	Hadoop	Spark	average	average	average
PEFT (FlexLog)	**0.718**	**0.784**	**0.723**	**0.642**	**0.682**	**0.672**	**0.704**	**0.982**	**0.892**	**0.937**	**0.972**	**0.971**
Few-shot ICL	0.519	0.588	0.565	0.550	0.593	0.487	0.550*	0.925	0.772	0.848*	0.876*	0.824*

* FlexLog yields a significant higher F1 score compared to the alternative learning strategy for Mistral (Few-shot ICL).