Empirical Software Engineering (2024) 29:105
https://doi.org/10.1007/s10664-024-10501-4

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

Fatemeh Hadadi[1] · Joshua H. Dawes[4] · Donghwan Shin[5] · Domenico Bianculli[4] · Lionel Briand[1,2,3]

Accepted: 17 May 2024 / Published online: 20 June 2024

Abstract
With the increasing complexity and scope of software systems, their dependability is crucial. The analysis of log data recorded during system execution can enable engineers to automatically predict failures at run time. Several Machine Learning (ML) techniques, including traditional ML and Deep Learning (DL), have been proposed to automate such tasks. However, current empirical studies are limited in terms of covering all main DL types—Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), and transformer—as well as examining them on a wide range of diverse datasets. In this paper, we aim to address these issues by systematically investigating the combination of log data embedding strategies and DL types for failure prediction. To that end, we propose a modular architecture to accommodate various configurations of embedding strategies and DL-based encoders. To further investigate how dataset characteristics such as dataset size and failure percentage affect model accuracy, we synthesised 360 datasets, with varying characteristics, for three distinct system behavioural models, based on a systematic and automated generation approach. Using the F1 score metric, our results show that the best overall performing configuration is a CNN-based encoder with Logkey2vec. Additionally, we provide specific dataset conditions, namely a dataset size $>$ 350 or a failure percentage $>$ 7.5%, under which this configuration demonstrates high accuracy for failure prediction.

Keywords Logs · Failure prediction · Deep learning · Embedding strategy · Synthesised data generation · Systematic evaluation

1 Introduction

As software systems continue to increase in complexity and scope, reliability and availability play a critical role in quality assurance and software maintenance (Bauer and Adams 2012; Le and Zhang 2022). During runtime, software systems often record log data about their execution, designed to help engineers monitor the system's behaviour (He et al. 2021). One

Communicated by: Lei Ma

Extended author information available on the last page of the article 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 (Carvalho et al. 2019).

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 (Das et al. 2020, 2018; Sahoo et al. 2003) and anomaly detection (Du et al. 2017; Meng et al. 2019; Zhang et al. 2019), have been proposed over the past few years. Machine Learning (ML) has played a key role in automatic log analysis, from Traditional ML methods (e.g., Random Forest (RF) (Breiman 2001), Support Vector Machine (SVM) (Cortes and Vapnik 1995), Gradient Boosting (GB) (Chen et al. 2019)) to Deep Learning (DL) methods (e.g., DeepLog (Du et al. 2017), LogRobust (Zhang et al. 2019), LogBERT (Guo et al. 2021)) relying on various DL network architectures, including Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), and transformers (Le and Zhang 2022).

Although several studies have explored the use of DL models with various log sequence embedding strategies (He et al. 2021), 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 and Zhang (2022) and Lu et al. (2018) 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 (He et al. 2021). 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 paper, 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 (Devlin et al. 2018)), a template ID-based strategy Logkey2vec (Lu et al. 2018), and aggregation of semantic and template ID-based strategies, FastText with TF-IDF (Zhang et al. 2019), are considered. In the classification step, four types of DL models, including LSTM (Hochreiter and Schmidhuber 1997a), BiLSTM (Schuster and Paliwal 1997), CNN (O'Shea and Nash 2015), and transformer (Vaswani et al. 2017).

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 (Shin et al. 2022; Walkinshaw et al. 2013) to available system logs. When synthesizing 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_PF, to compare the results obtained on synthesized 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 conclude that the best model includes the CNN-based encoder with Logkey2vec as an 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 in terms of dataset size and failure percentage. Our findings provide valuable insights for software and AIOps engineers to select the best DL-based solution for optimal failure prediction. Moreover, we aim to provide guidance in optimising dataset characteristics to improve failure prediction accuracy. In conclusion, this paper offers clear guidelines for those looking to leverage DL in predicting system failures from logs.

To summarise, the main contributions of this paper are:

- A large-scale, systematic investigation of the application of various DL encoders—LSTM-, BiLSTM-, CNN-, and transformer-based—and embedding strategies—BERT (Devlin et al. 2018), Logkey2vec (Lu et al. 2018) and hybrid strategy combining FastText with TF-IDF (Zhang et al. 2019)—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 synthesized 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 a best-performing traditional ML-based failure predictor 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 paper is organised as follows. Section 2 presents the basic definitions and concepts that will be used throughout the paper. Section 3 illustrates related work. Section 4 describes the architecture of our failure predictor with its different configuration options. Section 5 describes our research questions, empirical methodology, and synthetic log data generation. Section 6 reports empirical results. Section 7 discusses the implications of the results. Section 8 concludes the paper and suggests future directions for research and improvements.

2 Background

In this section, we provide background information on the main concepts and techniques that will be used throughout the paper. First, we briefly introduce the concepts related to finite state automata (FSA) and regular expressions in § 2.1 and execution logs in § 2.2. We then describe two important log analysis tasks (anomaly detection and failure prediction) in § 2.3 and further review machine-learning (ML)-based approaches for performing such tasks in § 2.4. We conclude by providing an overview of embedding strategies for log-based analyses in § 2.5.

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 $\delta \colon Q \times \Sigma \to Q$ is the transition function. The extended transition function $\delta^* : Q \times \Sigma^* \to Q$, 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 Logs

In general, a *log* is a sequence of log messages generated by logging statements (e.g., `printf()`, `logger.info()`) in the source code (He et al. 2021). A *log message* is textual data composed of a *header* and *content* (He et al. 2021). 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.

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

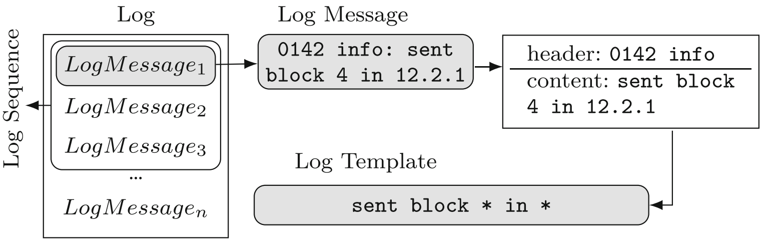


Fig. 1 An example illustrating the concepts of log, log message, log template, and log sequence

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 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`.

2.3 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 (He et al. 2021); we provide an overview of these tasks below.

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 (He et al. 2021), indicating possible errors, faults, or failures in software systems.

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 and Zhang (2022) 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, each partition usually being labelled as an anomaly either when an error, unknown, or failure message appears in 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 (Bogatinovski et al. 2022), as compared in Fig. 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 analyzed. 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.

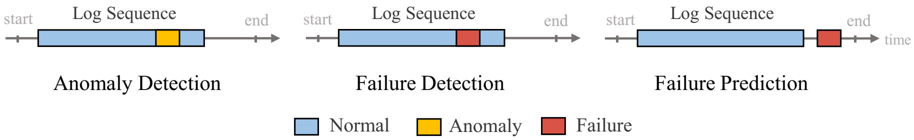


Fig. 2 Illustration of Log Analysis Tasks

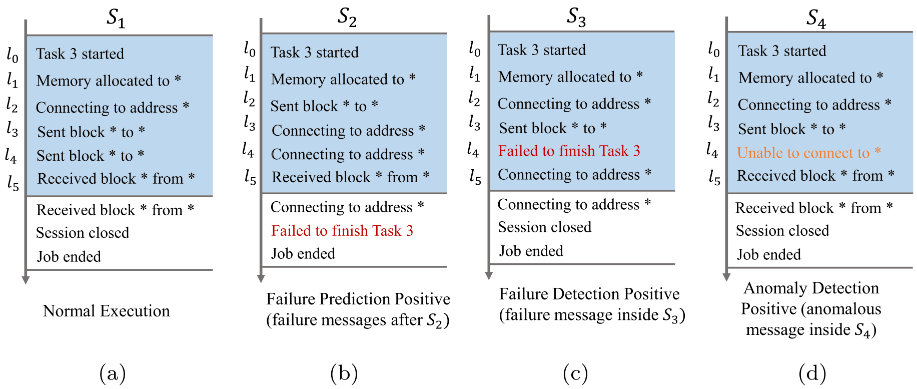
2.3.2 Failure Prediction

Failure prediction attempts to proactively generate alerts *before* the occurrence of failures, which often lead to unrecoverable outages (He et al. 2021). 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 Fig. 2, anomaly or 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 of failure prediction typically consists of normal-looking inputs, a subset of which involves subtle and complex patterns in logs, which may be associated with a future failure. Patterns can indicate impending issues that have not yet manifested as failures in log data.

Figure 3 shows a simplified comparison of "positive sequences" (in contrast to "normal" sequences) for the aforementioned tasks (depicted in Subfigures 3b, c and d), next to a normal log (depicted in Subfigure 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 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 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


S_1

l_0 Task 3 started
l_1 Memory allocated to *
l_2 Connecting to address *
l_3 Sent block * to *
l_4 Sent block * to *
l_5 Received block * from *
Received block * from *
Session closed
Job ended

Normal Execution

(a)

S_2

l_0 Task 3 started
l_1 Memory allocated to *
l_2 Sent block * to *
l_3 Connecting to address *
l_4 Connecting to address *
l_5 Received block * from *
Connecting to address *
Failed to finish Task 3
Job ended

Failure Prediction Positive (failure messages after S_2)

(b)

S_3

l_0 Task 3 started
l_1 Memory allocated to *
l_2 Connecting to address *
l_3 Sent block * to *
l_4 Failed to finish Task 3
l_5 Connecting to address *
Connecting to address *
Session closed
Job ended

Failure Detection Positive (failure message inside S_3)

(c)

S_4

l_0 Task 3 started
l_1 Memory allocated to *
l_2 Connecting to address *
l_3 Sent block * to *
l_4 Unable to connect to *
l_5 Received block * from *
Received block * from *
Session closed
Job ended

Anomaly Detection Positive (anomalous message inside S_4)

(d)

Fig. 3 Comparison of Normal Sequence (on the left) and Positive Sequences in Log Analysis Tasks (on the right)

anomaly detection, an anomalous log message, highlighted in yellow, appears within S_4 (see Subfigure 3d).

Dataset Transferability for Failure Prediction It is worth mentioning that, as sketched in Subfigure 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 intended specifically for failure detection in the context of failure prediction, some assumptions should hold. First, log data is required to be ordered by timestamp, to make it possible to separate the sequence of messages before the occurrence of a failure. 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 § 5.2.3).

2.4 DL Techniques in Log Analysis

In recent years, a variety of deep learning (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 and Zhang (2022), 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 (Du et al. 2017; Das et al. 2018; Zhang et al. 2019). We now explain the major features of each category as well as their variations.

2.4.1 RNN

LSTM (Hochreiter and Schmidhuber 1997b; Gers et al. 2000) is an RNN-based model commonly used in both anomaly detection and failure prediction (Du et al. 2017; Das et al. 2018). An LSTM network consists of multiple units, each of which is composed of a cell, an input gate, an output gate (Hochreiter and Schmidhuber 1997b), and a forget gate (Gers et al. 2000). 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 (Meng et al. 2019, 2020; Das et al. 2020, 2018).

BiLSTM is an extension of the traditional LSTM (Huang et al. 2015). 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 is able to capture complex dependencies and produce more accurate predictions. The BiLSTM-based model has achieved accurate results for anomaly detection (Zhang et al. 2019).

2.4.2 CNN

CNN is a neural network primarily employed for image recognition (O'Shea and Nash 2015). 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 (2014), 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 there is no padding added to the input sequence, leading to feature maps smaller than the input sequence. Padding is a technique employed to add zeros to the beginning and/or end of the sequence; it allows for more space for the filter to cover, controlling the size of output feature maps. Padding is commonly used so that the output feature map has the same length as the input sequence (Chollet 2017).

The pooling layer reduces the spatial dimensions of the feature maps extracted by the convolutional layer and simplifies the computational complexity of the network.

Recently, CNNs have shown high-accuracy performance in anomaly detection (Lu et al. 2018).

2.4.3 Transformer

The transformer is a type of neural network architecture designed for natural language processing tasks, introduced by Vaswani et al. (2017). 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 with respect to the 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 (Le and Zhang 2021; Huang et al. 2020; Guo et al. 2021; Nedelkoski et al. 2020), showing outstanding performance.

2.4.4 GNN

A Graph Neural Network (GNN) is a neural network designed to process data structured as graphs (Dwivedi et al. 2021). During training, taking a graph-structured input, it updates node feature vectors (where nodes are equivalent to vertices in the graph) iteratively with respect to feature vectors of its neighbour 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 is the data structure they process. GNNs process data structured as graphs. Since log data are initially sequential data, it requires further processing to construct a graph from sequential data. 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 (Dwivedi et al. 2021) to combine node feature vectors.

We note that GNNs are sometimes regarded as a representation method, such as log sequence embedding strategies detailed in § 2.5, since they compute the graph representation of log sequences (Wu et al. 2023). However, we consider them as a classification method, like the other DL methods described in § 2.4, since current GNNs necessitate pre-existing semantic embeddings for input.

2.5 Log Sequence Embedding Strategies

When analyzing log sequences, the textual data of log sequences' elements must be converted into a vector representation that is understandable by a machine; such a 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 (Deerwester et al. 1990), (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.

2.5.1 Template ID-based Strategy

There are many studies that have achieved high accuracy results by using log embedding strategies that rely on the ID numbers or count vectors of log sequence elements (Chen et al. 2021). 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 (Rajaraman et al. 2014) 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 it incompatible with methods requiring an embedding vector at the log template level.

Logkey2vec, introduced by Lu et al. (2018), is another strategy used in log analysis, which is based on log template IDs and is able to transform 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 by random numbers and are improved 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 linguistics 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 (Mikolov et al. 2013) which is pre-trained using CBOW (Continuous Bag-of-Words) and Skip-grams (Mikolov et al. 2013), 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 et al. (2022) (who used the term “vectorizer”), and by Guo et al. (2021) (who used the term “Embedding Matrix”).

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 et al. (2019) proposed *template2vec*, an embedding strategy based on 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 the contextual information of text into account. It has been used in a few studies in log sequence embedding (Guo et al. 2021; Le and Zhang 2021). This model fares better than the other pretrained transformer-based models: GPT2 (Radford et al. 2019) and RoBERTa (Liu et al. 2019) in log sequence embedding (Le and Zhang 2021).

The pre-trained BERT base model (Devlin et al. 2018) 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 inside 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 (Wu et al. 2016), which is able to 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 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.

2.5.3 Hybrid Strategy

This category aims to combine the benefits of both template ID-based and semantic-based strategies while compensating for their limitations. The main instance of this category is the study of Zhang et al. (2019). They leverage FastText Joulin et al. (2016) 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 (Foundation 2023); 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 one vector representing a log template using a weighted average with TF-IDF (calculated 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 (1), where $w_i \in R$ indicates the TF-IDF value of t_i.

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

This strategy seeks to retain the advantages of the previous strategies. If a word is frequently mentioned among log templates, it is given a lower TF-IDF weight during the aggregation of word vectors, increasing the distinction of embedding vectors between log templates. Moreover, similar to BERT but not as informative in terms of word context, FastText assigns vectors with high cosine similarity to two log templates that contain different words but are semantically close.

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.

3.1 Related Empirical Studies

3.1.1 Log-based Anomaly Detection

As discussed in § 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 (Das et al. 2018, 2020). There are several papers reporting 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 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 § 2.5, in the next column;

Table 1 Overview of Related Empirical Studies

Paper	DL Type(s)	LSE Strategi(es)	Dataset(s)	Control of Dataset Characteristics	Labelling Scheme
Anomaly Detection					
Lu et al. (2018)	LSTM, CNN, MLP	Logkey2vec	HDFS	No	L
Meng et al. (2019)	LSTM	template ID Template2Vec	HDFS, BGL	No	T, L
Huang et al. (2020)	LSTM, BiLSTM, Transformer	count vector, F+T, Log Encoder	HDFS, BGL, OpenStack	Yes (unstable log injection ratio)	T, L
Yang et al. (2021)	LSTM, BiLSTM,GRU	template ID,TF-IDF, F+T	HDFS, BGL	No	T, L
Guo et al. (2021)	LSTM, Transformer	template ID, count vector, Embedding Matrix	HDFS, BGL, Thunderbird	No	T, L
Le and Zhang (2021)	LSTM, BiLSTM, Transformer	count vector, Log2Vec*, F+T, BERT	HDFS, BGL, Spirit, Thunderbird	No	T, L
Bogatinovski et al. (2022)	LSTM, Transformer	count vector, vectorizer	OpenStack_v2	Yes (unstable log injection ratio)	T
Le and Zhang (2022)	LSTM, BiLSTM, GRU, CNN	template ID, Logkey2vec, F+T	HDFS, BGL, Spirit, Thunderbird	Yes (class distribution, data noise, partitioning methods)	T, L
Xie et al. (2022)	BiLSTM, CNN Transformer, GNN	count vector, Logkey2vec, F+T, BERT	HDFS, BGL, Spirit, Thunderbird	Yes (partitioning methods)	T, L
Wu et al. (2023)	MLP, CNN LSTM	count vector, TF-IDF, Word2Vec, FastText, BERT	HDFS, BGL, Spirit, Thunderbird	Yes (partitioning methods)	T, L
Failure Prediction					
Lin et al. (2018)	BiLSTM	N/A	AzureML	No	T
Das et al. (2018, 2020)	LSTM	template ID	Clay-HPC	No	L
Our Study**	LSTM, BiLSTM,	Logkey2vec,	Synthesized Data,	Yes (Dataset size, Failure	L
CNN, Transformer	BERT, F+T	OpenStack_FP	Percentage, LSL, Failure Pattern type)		

**: we highlight that Log2Vec is different than Logkey2vec, a log sequence embedding strategy (see* § 2.5.1)
***: further discussed in* § 7

notice there are a few models not using LSE, such as DeepLog (Du et al. 2017). 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, based either on a log identifier or on timestamp (represented by *L* and *T*, respectively).

We now briefly explain the included papers with the aim of motivating our study and highlighting 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 et al. (2018) introduced CNN for anomaly detection as well as the Logkey2vec embedding strategy (see § 2.5.1). They compared it to LSTM and MLP networks, also relying on the Logkey2vec embedding strategy. Meng et al. (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 et al. (2020), featuring three DL models: HitAnomaly (transformer-based), LogRobust (Zhang et al. 2019) (BiLSTM-based), and DeepLog (LSTM-based). HitAnomaly utilises transformer blocks (see § 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 et al. (2021) proposed the GRU-based (Cho et al. 2014) PLELog and compared it to LogRobust and DeepLog. PLELog used the TF-IDF technique and LogRobust used F+T. Guo et al. (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 (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 § 2.5.2) for Neurallog and Log2Vec (Meng et al. 2020) (a strategy based on Word2Vec) for DeepLog.

An important recent work on failure detection is the study of Bogatinovski et al. (2022). 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 (Yamanishi and Maruyama 2005). 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 and Zhang (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 et al. (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 et al. (2022) took into account a wide range of DL techniques and LSE strategies from each category, their results, similar to those of Le and Zhang (2022), were obtained using only public datasets.

Finally, (Wu et al. 2023) studied the effectiveness of different LSE on ML-based models for anomaly detection. In contrast to the study of Le and Zhang (2022), 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. Similarly to Le and Zhang (2022), 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 et al. (2022), 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.

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 et al. (2003) 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 et al. (2015) 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 et al. 2018) 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 et al. (2018) 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 et al. 2020) 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 et al. (2018) 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 (Das et al. 2020), as state-of-the-art with faster inference time. Both Dash and Aarohi rely on the template ID-based strategy for embedding (see § 2.5). The above DL-based studies of failure prediction are briefly summarised in Table 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 Das et al. (2020) and Dash (Das et al. 2018). On the other hand, the Prefix dataset is available but is of limited use, due to its low complexity leading easily to high accuracy regardless of the approach. As a result, We found the limited number of publicity available datasets to be a hindrance. We therefore opted to develop a method for synthesising new datasets, as described next.

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 (Li et al. 2020), or evaluation of logs driven from system updates (Zhang et al. 2019; Huang et al. 2020). 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 (Blom et al. 2005; Bombarda and Gargantini 2020). Given the restrictions of available and suitable datasets for our failure prediction problem (as discussed in § 2.3), we henceforth refer to the second group of algorithms when mentioning data synthesis.

In 2005, Blom et al. (2005) proposed a method for generating test suites for systems whose behaviours can be described by extended finite state machines (EFSM). This method 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 et al. (2017) introduced EMSBench, which contains a model capable of mimicking complex system behaviour. Using this model, sequential traces are generated for the purpose of comparing different platforms. In 2020, Bombarda and Gargantini (2020) 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, Krstić and Schneider (2020) presented an algorithm for generating an event stream with their associated arbitrary values. These logs are compatible with system specifications in the first-order dynamic logic (MFODL) (Basin et al. 2020). We will further discuss existing data synthesis algorithms and their differences with ours when we present the latter in § 5.3.

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 from the description of the empirical study design (see Sect. 5).

Figure 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 § 4.1.

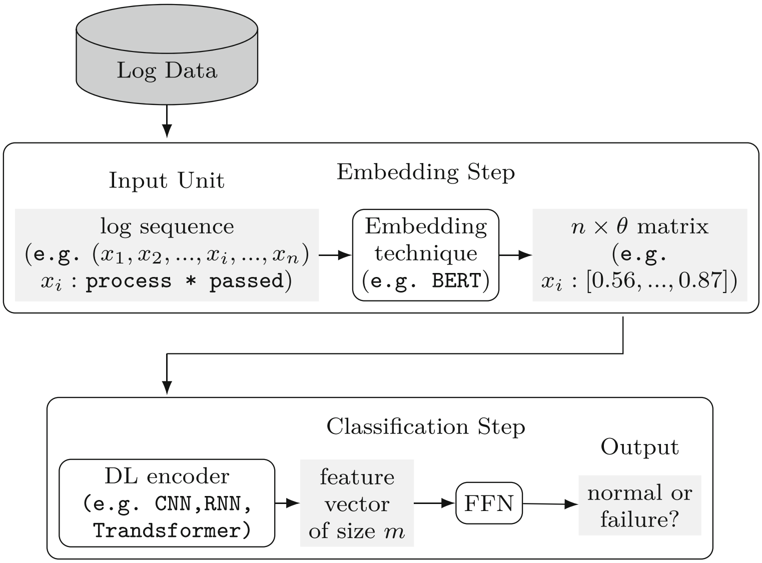


Fig. 4 Overview of the modular architecture for failure prediction

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 § 4.2. Similar to related studies (Huang et al. 2020; Lu et al. 2018), 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 the one of the NeuralLog model (Le and Zhang 2021) as well as the LogRobust one (Zhang et al. 2019).

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.

$$o = rW_2 + b_2 \tag{2}$$

$$softmax(o_p) = \frac{exp(o_p)}{\sum_j exp(o_j)} \tag{3}$$

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 (Vaswani et al. 2017). This language model handles the log templates as well as learning the language of failure patterns to predict the label of sequences.

To train the above architecture, a number of hyper-parameters should be set such as the choice of the optimizer, loss function, learning rate, input size (for some deep learning models), batch size, and the number of epochs. Tuning these hyper-parameters is highly recommended as it significantly increases the chances of achieving the best failure prediction accuracy. Section 5.2.4 will detail the training and hyper-parameter 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.

4.1 Embedding Strategies

While the modular architecture can accommodate various log sequence embedding options, we only consider one representative instance from each of three LSE strategies (see § 2.5), given our experimental constraint. More details are provided in § 5.2.1.

Note that following three techniques were not compared in the same study before, according to Table 1.

Logkey2vec For Logkey2vec (see § 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 et al. (2018).

BERT The maximum number of input tokens for BERT (see § 2.5.2) is 512 tokens. This limit does not constitute a problem in this work since 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, there are related studies suggesting approaches to use BERT accordingly (Weijie et al. 2021; Ding et al. 2020; Sun et al. 2019). 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 (Zhang et al. 2019), the dimension of the embedding vector is set to 300 ($d = 300$).

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 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 (Xie et al. 2022).

LSTM-based This DL model is inspired by the LSTM architecture suggested by related works, including DeepLog (Du et al. 2017), Aarohi (Das et al. 2020), and Dash (Das et al. 2018). The model contains one LSTM hidden layer with 128 nodes and ReLu activation. A

Dropout with a 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 (Zhang et al. 2019).

CNN-based The CNN architecture is a variation of the convolutional design for the CNN-based anomaly detection mode (Lu et al. 2018). Based on our preliminary experimental results, 20 filters, instead of one, for each of the three 1D convolutions (see § 2.4.2) are used in parallel 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 (Le and Zhang 2021; Huang et al. 2020; Nedelkoski et al. 2020). The model is composed of two main parts: positional embedding and transformer blocks. One transformer block is adopted after positional embedding, set similarly to a recent study (Le and Zhang 2021). 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 § 2.4.

5 Empirical Study Design

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 configuration 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 § 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 answer to these questions is expected to lead to practical guidelines for choosing the best failure prediction model given a dataset with certain 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 additionally investigates the impact of the input dataset characteristics on failure prediction accuracy with a focus on the best DL encoder and log sequence embedding strategy found 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 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 § 5.2.3).

5.2 Methodology

As discussed in § 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 training datasets' characteristics (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 failure predictor training and testing 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 since, according to the comprehensive study of Fernández-Delgado et al. (2014), it has shown the best performance overall 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 and SVM (Zhang et al. 2018; Liu et al. 2020) (see also § 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 hyper-parameter, to 10, in line with its related study (Wu et al. 2023). For 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 (Wu et al. 2023).

To answer RQ4, we first identify all the top configurations since there might be certain datasets where configurations other than the best configuration inferred from RQ1-2 fare 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 (Breiman et al. 1984a) 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 § 3.1.2), we must choose from the datasets available 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 § 2.3.2. OpenStack is a common dataset explicitly used for failure prediction (Bogatinovski et al. 2022) and further labelled at the log message level that we refer to as OpenStack_v2. These characteristics allowed us to further process it to make it suitable for failure prediction, leading to the creation of a new dataset called OpenStack_PF, which we introduce in § 5.2.3. We compare the failure prediction accuracy results obtained on the synthesized datasets most similar, in terms of dataset size, failure percentage, and MLSL [1] to OpenStack_PF, with those obtained on the OpenStack_FP dataset. For practical reasons, we only focus on the accuracy results of the best DL configuration as well as the best traditional ML model, i.e., RF.

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 shown to be accurate in the literature as discussed in § 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 § 2.5). TF-IDF methods (like count vectors) were incompatible with our architecture since 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 Sect. 2.5.2. Among available pre-trained strategies, we included BERT, given its prevalent usage in log analysis studies and its demonstrated benefits (Guo et al. 2021; Le and Zhang 2021). As for the hybrid strategy, we included F+T (aggregation of FastText with TD-IDF), which is a common hybrid strategy 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 § 4.2. We configured the encoders based on the recommendations reported in the literature (see § 4.2 for further details).

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 (Le and Zhang 2022): 200, 500, 1000, 5000, 10000, and 50000.

The LSL could affect failure prediction since a failure pattern that spans a longer log might be more difficult to predict correctly. Similar to observed lengths in real-world log sequences

[1] More details are provided in § 6.5.

across publicly available datasets (Le and Zhang 2022), we vary the maximum[2] LSL across five levels: 20, 50, 100, 500, and 1000.

The failure percentage determines the balance of classes in a dataset, which may affect the performance of DL models (Johnson and Khoshgoftaar 2019). 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 (Lin et al. 2018). 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, failure patterns defined in previous studies are too simple; for example, (Das et al. 2018) consider a specific, consecutive sequence of problematic log templates, called a "failure chain". But in practice, not all problematic 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. 2021) 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 § 5.3.

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 et al. (2019). It is known to be the most comprehensive publicly available dataset of logs including failure data generated from a cloud-based system (Bogatinovski et al. 2022), 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

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

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

the labels is at the level of the workload; labels are determined by checking assertions at the end of the workload runs. Bogatinovski et al. (2022) further labelled the logs at the log message level using two human annotators labelling more than 200000 log messages to find those indicating the logged failure. We name this version of the dataset OpenStack_v2. To make OpenStack_v2 ready for failure prediction, we further processed it according to the discussion on dataset transferability, as mentioned in § 2.3.2.

Specifically, we partition the logs according to their log identifier, which is the task ID in this context. As discussed in § 2.3.1, partitioning logs using log identifiers leads to 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 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 maximum log sequence length, in case a log sequence exceeds the limit, we only keep the last 1000 log messages. We set this threshold since it is the maximum input sequence length in our modular architecture; moreover, we speculate the messages at the end of the sequence to be more related to the subsequent failure. We name the processed dataset OpenStack_PF, as it is suitable for failure prediction. Table 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.

5.2.4 Failure Predictor Training and Testing

We split each artificially generated dataset, as well as OpenStack_PF, 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 (Prechelt 1998) during training to avoid over-fitting.

For training failure predictors, to control the effect of highly imbalanced datasets, oversampling (Upton and Cook 2008) 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. For all the training datasets, we use the Adam optimizer (Kingma and Ba 2015) with a learning rate of 0.001 and the sparse categorical cross-entropy loss function (Chen et al. 2022) 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 keep the training time reasonable without significantly affecting training effectiveness. Specifically, we set the two hyperparameters as follows:

Table 2 Overview of OpenStack_PF dataset

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

- *Batch size*: By default, we set it to 10, 15, 20, 30, 150, and 300 for dataset sizes of 200, 500, 1000, 5000, 10000, and 50000, respectively. If the failure percentage is less than or equal to 30 (meaning more oversampling will happen to balance between normal and 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 1000, 5000, 10000, and 50000, respectively, to reduce training time.

Table 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 1000, 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 with cloud computing environments provided by the Digital Research Alliance of Canada (Digital Research Alliance of Canada 2016), on the Cedar cluster with a total of 94528 CPU cores for computation and 1352 GPU devices.

5.3 Synthetic Data Generation

In defining a set of factors, the methodology described in § 5.2 makes it clear that there is a need for a mechanism that can generate datasets in a controlled, unbiased manner. Recent works on data synthesis have used finite-state automata (§ 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 (§ 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 § 5.2, such as *LSL*, the case for a mechanism for automated, controlled generation of datasets becomes yet stronger.

Table 3 Overview of Hyperparameter Setting

Hyperparameter	Condition	Dataset Size					
		200	500	1000	5000	10000	50000
Batch Size	Default	10	15	20	30	150	300
	PF $\leq$ 30	10	15	30	60	300	600
	MLSL $\geq$ 500*	5	5	5	5	5	5
Number of Epochs	Default	20	20	20	20	20	20
	MLSL $\geq$ 500	20	20	10	10	5	5

* This condition has higher priority than the other

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 § 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 are applicable 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).

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 (Shin et al. 2022; Walkinshaw et al. 2013) 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, 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.

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 § 2.1.

A behaviour model has the particular characteristic that its alphabet Σ consists of *log template IDs* (see § 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 need 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). 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 § 2.1). 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 $\mathcal{M}$.

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* (Tauber 2018) 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 with respect to the range of dataset size (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) (*maximum LSL*, see § 5.2). The Python package *random* (Package 2019) 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 § 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 a random selection of choices on the RegEx's parse tree nodes. *Random* package uses the Mersenne twister algorithm (Matsumoto and

Nishimura 1998) to generate a uniform pseudo-random number used for random selection tasks.

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 § 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 § 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 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 : $\langle s_1, s_2, \ldots, s_n \rangle \in \mathcal{L}(\mathcal{M})$
sequence : list ← filteredRandomWalk($\mathcal{M}$, mlsl)
while sequence $\in \bigcup_{fp_i \in \mathsf{failurePatterns}(\mathcal{M})} \mathcal{L}(fp_i)$ do
    sequence ← filteredRandomWalk($\mathcal{M}$, mlsl)
return sequence
```

In particular, Algorithm 1 generates a normal log sequence by:

1. Generating a random log sequence by random walk (invoking Algorithm 2);
2. Looking for a failure pattern $fp \in \mathsf{failurePatterns}(\mathcal{M})$ that matches the generated log sequence;
3. Repeating until a log sequence is generated that matches no failure pattern.

Ultimately, Algorithm 1 is relatively lightweight; the weight of the work is performed by Algorithm 2, which we now describe in detail.

The input arguments of Algorithm 2, which defines the procedure *filteredRandomWalk*, are a behaviour model $\mathcal{M}$ and the maximum LSL, *mlsl*.

The algorithm proceeds as follows. First, on line 1, we invoke the *calculateSValues* function to compute a map that sends each state $s \in Q$ of $\mathcal{M}$ to the length of the shortest path from that state to an accepting state in A. Next, on line 2, the *sequence* variable is initialised to an empty sequence. As the algorithm progresses, this variable stores the generated sequence of log template IDs. To help with this, the variable *currentState* is initialised to keep track of

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  $sValue : int \leftarrow calculateSValues(\mathcal{M})$
2  $sequence : list \leftarrow \langle\rangle$
3  $maximumWalk : int \leftarrow mlsl$
4  $currentState : state \leftarrow q_0$
5  while $currentState \notin A$ do
6      $options : set \leftarrow \emptyset$
7      $transitions : set \leftarrow \{\langle currentState, s, q \rangle : \delta(currentState, s) = q\}$
8      for $\langle q, s, q' \rangle \in transitions$ do
9          if $sValue(q') < maximumWalk$ then
10             $options \leftarrow options \cup \{\langle q, s, q' \rangle\}$
11     $\langle q, s, q' \rangle \leftarrow$ random choice from options
12     $sequence \leftarrow sequence + \langle s \rangle$
13     $currentState \leftarrow q'$
14     $maximumWalk \leftarrow maximumWalk - 1$
15 return sequence
```

the state that the algorithm is *currently in* during the walk of the behavioural model. Hence, this variable is initialised on line 4 as the initial state. The final step in the setup stage of our algorithm is to initialise the *maximumWalk* variable, which serves as a counter to ensure the limit on the length of the generated log sequence (defined by *mlsl*) is respected.

In the while loop (line 5), as long as the current state, *currentState*, is not yet an accepting state, the random walk transits from the current state to a new state. The set of possible transitions to take is computed by line 7, and stored in the variable *transitions*. Each transition is represented by a triple containing the starting state, the symbol to be observed, and the state resulting from observation of that symbol. Once this set has been computed, the algorithm performs a filtering step. In particular, in order to ensure that we respect the limit imposed on the length of the generated path by *mlsl*, we only consider transitions that lead to a state q' such that $sValue(q') < maximumWalk$. The resulting list of valid options is then held in the variable *options*.

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.

While Algorithm 2 generates an unlabelled log sequence, Algorithm 1 generates a normal log sequence. To do this, it starts by generating a log sequence, by invoking 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 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.

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 Fig. 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 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.

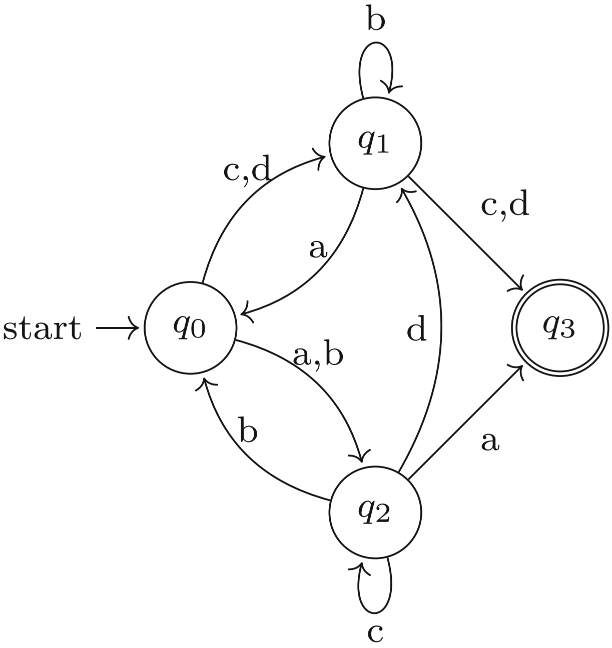


Fig. 5 An example of a behaviour model

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.

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.

Table 4 s values for each state

state	s value
q_0	2
q_1	1
q_2	1
q_3	0

5.3.7 Compliance to Requirements

We now describe how the approach that we have described meets the remaining requirements set out in § 5.3.1.

R1 is met because we have two procedures for generating failure log sequences (§ 5.3.4) and normal log sequences (§ 5.3.5). By having these procedures, we can precisely control the number of each type of log sequence in our dataset.

R4 is met because of the randomisation used in our data generation algorithm, described in Sects. 5.3.4 and 5.3.5.

5.4 Experimental Setting for Synthesised Data Generation

To generate diverse log datasets with the characteristics described in § 5.2.2, using the syntactic data generation approach described in § 5.3, we need two main artifacts: *behaviour models* and *failure patterns*.

5.4.1 Behaviour Models

Regarding behaviour models, as discussed in § 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 (Walkinshaw et al. 2013) and PRINS (Shin et al. 2022). 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 et al. (2022):

(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 § 5.2.2;

(2) Since there is no straightforward way of automatically generating failure patterns for individual behaviour models considering the two failure pattern types, we had to manually generate failure patterns (detailed in § 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 1000);

(3) The model inference time should be less than 1 h; 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). Table 5 reports about 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) (Black 2020), which indicates the complexity of a behaviour model (the higher, the more complex).

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 § 5.3.3. Also, note that we need two types of

Table 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

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 § 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 1: 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 1: 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 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 § 5.3.4), there is a wide variability among failure patterns across each characteristic.

Table 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)

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:

- 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 allows us to see how our failure prediction machinery performs when the language of log sequences to work with is infinite vs. 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.

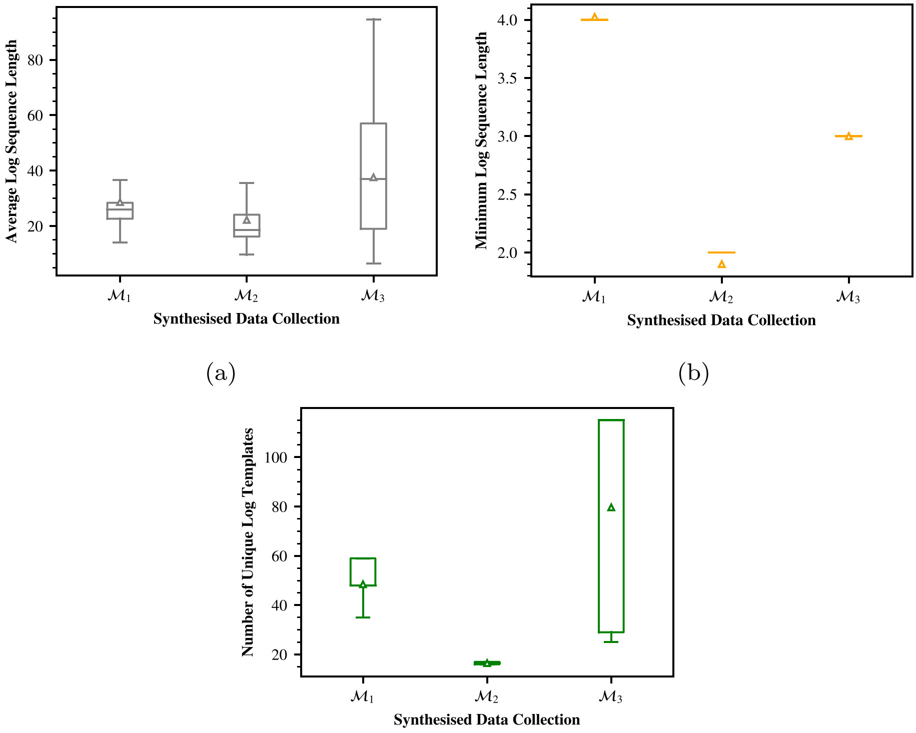
Finally, in varying how many (if any) Kleene stars are used, we control how many opportunities for *cycles* we have when traversing our behaviour model. For example, if we have a^*, then we have a single opportunity to loop on a. If we have $(a \mid b)^*(c \mid d)^*$, then we have two opportunities to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

5.4.4 Overview of Synthesised Data.

As the correctness of the synthetic data generation was discussed in § 5.3, the synthesised datasets should follow the desired characteristics we specified in § 5.2.2. Here we present an overview of additional statistics regarding the dataset generation. Figure 6 summarises three statistics in terms of average and minimum length of log sequences (Subfigure 6a and b) as well as the number of unique log templates in each dataset (Subfigure 6c). Each box is based on 360 datasets generated from its corresponding behavioural model $\mathcal{M}_1$, $\mathcal{M}_2$, or $\mathcal{M}_3$. We note that log sequences inside synthesised datasets are directly generated by our approach introduced in § 5.3. Thus, no partitioning method is needed. However, given that each log sequence simulates a complete walk (meaning from the initial state until the accepting state of a behavioural model), it more closely resembles the log sequences partitioned based on the log identifier. Based on Subfigure 6a, 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 § 5.4.1). Based on Subfigure 6b, the minimum length of log sequences remains relatively



(a)

(b)

(c)

Fig. 6 Overview of Synthesised Datasets

consistent across models. In Subfigure 6c, M1 and M2 closely align with the number of unique templates reported for $\mathcal{M}_1$ and $\mathcal{M}_2$ in Table 5. $\mathcal{M}_3$, 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. In this way, there may be some transitions in the finite automaton that are not 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.

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.

6.1 RQ1: DL Encoders

Figure 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) 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), 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 show poor results (low F1 score on average with very large 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 1000), the data-demanding transformer blocks are not well-trained, leading to poor performance. This limitation is thoroughly discussed in the literature (Xu et al. 2020).

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 § 2.4.1), it is still a recurrent network that has difficulties to remember a long input sequence (Lipton 2015). For this reason, since our log datasets contain long log sequences (up to a length of 1000), the LSTM-based encoder did not work well.

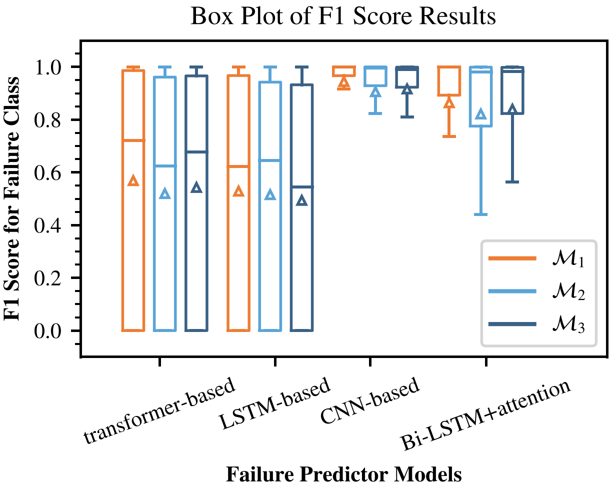


Fig. 7 Failure prediction accuracy for different DL encoders. The triangles additionally indicate mean values

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 which are associated with the failure pattern (Vaswani et al. 2017). 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 § 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 explained by 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 get 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 (Gu et al. 2018), 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.

6.2 RQ2: Log Sequence Embedding Strategies

Figure 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

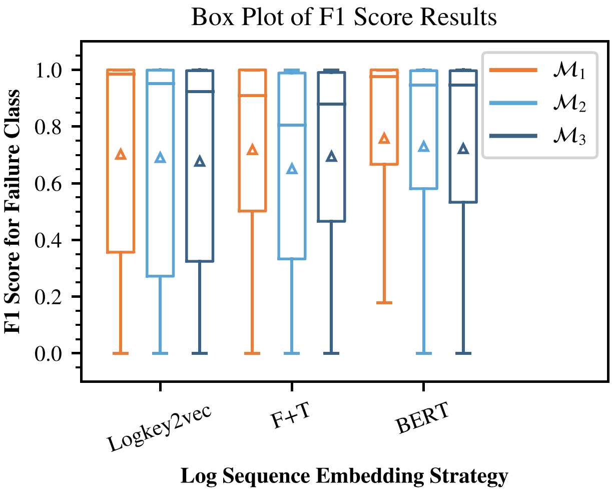


Fig. 8 Failure prediction accuracy for different log sequence embedding strategies. The triangles additionally indicate mean values

Table 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	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	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)

generated based on 360 × 4 data points since we have 360 combinations of dataset characteristics and four DL encoders. Similar to Fig. 7, the triangle in each box indicates the mean value. We now inspect the plots shown inside with the aim of answering our research questions. The plots based on precision and recall are excluded since they draw similar conclusions.

Figure 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 fares better than both F+T, which employs FastText but is not as informative as BERT, and Logkey2vec, which solely relies on log template IDs and does not account for the semantic information of templates.

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 7 reports the statistical test results. For example, the low p-value in column $\mathcal{M}_2$ 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).

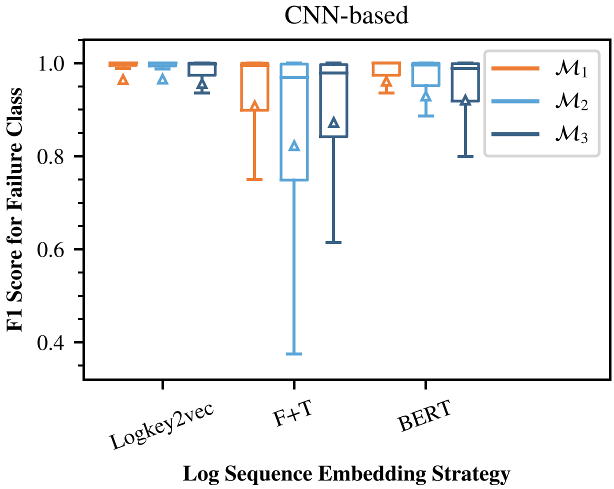


Fig. 9 Failure prediction accuracy of CNN-based model for different log sequence embedding strategies; triangles indicate mean values

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 § 6.1) and the LSTM-based encoder for $\mathcal{M}_3$. On the other hand, for the CNN-based encoder, the best overall embedding strategy is Logkey2vec, as clearly observable in Fig. 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 together with the DL encoder, while BERT is pre-trained independently from 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. 8, and similar rationales apply.

We note that BERT is still an attractive strategy for log sequence embedding when any other encoder 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.

6.3 RQ3: Traditional ML

Figure 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 (Fernández-Delgado et al. 2014; Zhang et al. 2018; Liu et al. 2020), 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 Fig. 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 § 2.4.2, 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 § 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-

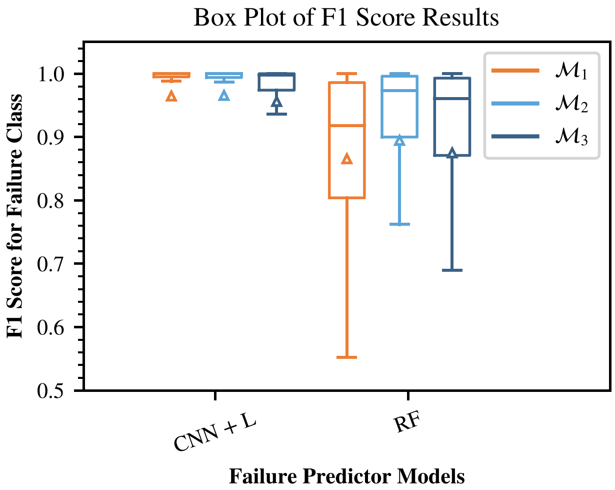


Fig. 10 Failure prediction accuracy of the best DL-based configuration (CNN with Logkey2vec) next to a traditional ML-based configuration (RF); triangles depict mean values

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 § 2.5.1.

> 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.

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

[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 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

BERT). Note that the top three configurations remained the same for different threshold values (r = 0, 0.05, 0.1). Table 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 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 § 6.6).

Figure 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

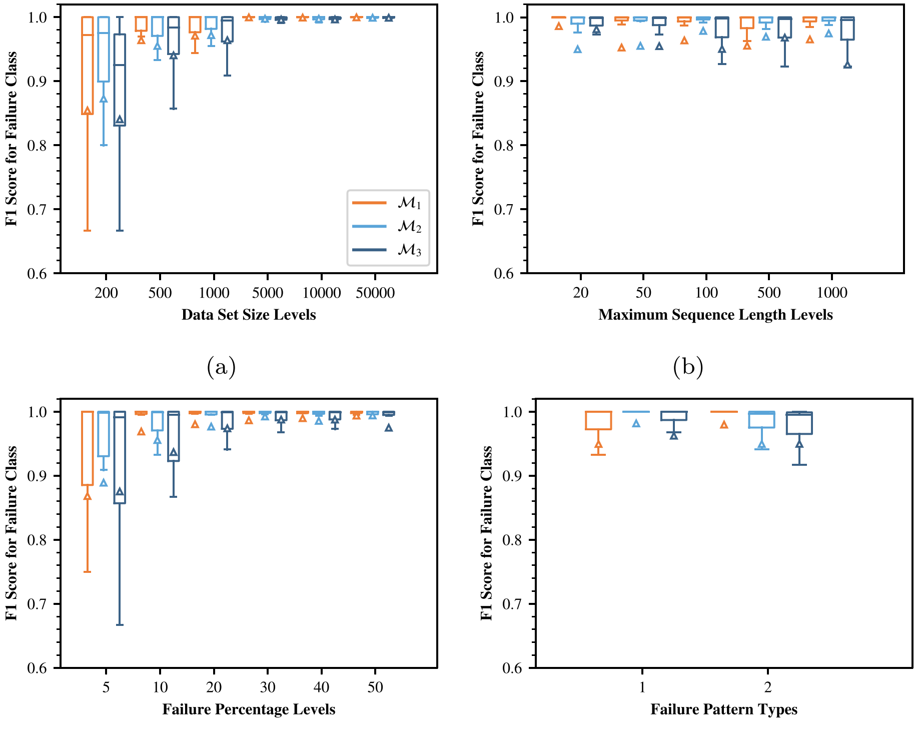


Fig. 11 Failure prediction accuracy of the CNN-based encoder with Logkey2vec for different dataset characteristics

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 Fig. 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 dataset size is 200, accuracy decreases below 0.7 in the worst case, whereas it always stays very close to 1.0 when dataset size is above or equal to 5000. Since larger datasets imply more training data, this result is intuitive but it clarifies data requirements for failure prediction.

Figure 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 1000. We suspect that the impact of log sequence length could be significant for much longer log sequences. However, log sequences longer than 1000 are not common in publicly available, real-world log datasets (Le and Zhang 2022) as explained in Sect. 5.2.2. Nevertheless, the investigation of much longer log sequences would be informative.

The relationship between failure percentage and failure prediction accuracy (F1 score) is depicted in Fig. 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 11d shows the failure prediction accuracy for different failure pattern types. There is no consistent trend across models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$; *Type-F* (the corresponding language is finite) is easier to detect than *Type-I* (the corresponding language is infinite) in $\mathcal{M}_2$ and $\mathcal{M}_3$, whereas the opposite happens in $\mathcal{M}_1$. It is unclear why, in $\mathcal{M}_1$, detecting less complex failure patterns (*Type-F*) is more difficult than detecting more complex patterns (*Type-I*). We may not have defined failure pattern types in a way that is conducive to explaining variations

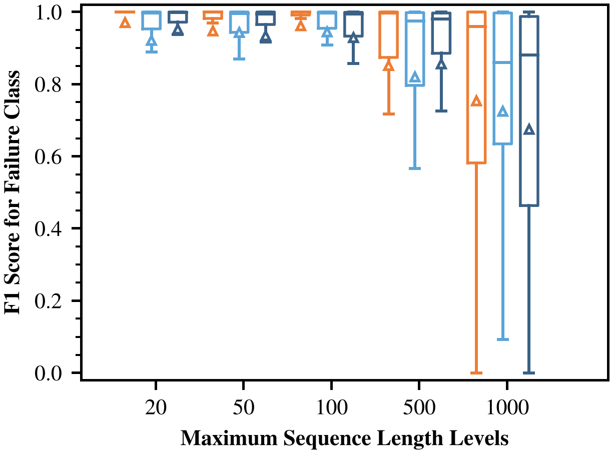


Fig. 12 Failure prediction accuracy of the BiLSTM-based encoder with BERT as a function of maximum sequence length

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 Fig. 12. Unlike CNN+L shown in Fig. 11(b), larger IQR and lower average values are clearly visible for longer log sequences in Fig. 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. To do this, we first labelled the 1080 datasets with the three best configurations; specifically, each dataset was labelled with the top performer among the three configurations. 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 (He and Garcia 2009) to address the class imbalance issue. We also performed Minimal Cost-Complexity Pruning (MCCP) (Breiman et al. 1984b) to avoid over-fitting. Figure 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 right-most leaf node indicates that CNN+L is the best configuration when the dataset size is larger than 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 predictor. 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 lower than or equal to 3000 and failure percentages lower than or equal to 15%, BiLSTM+B performs better than CNN+L

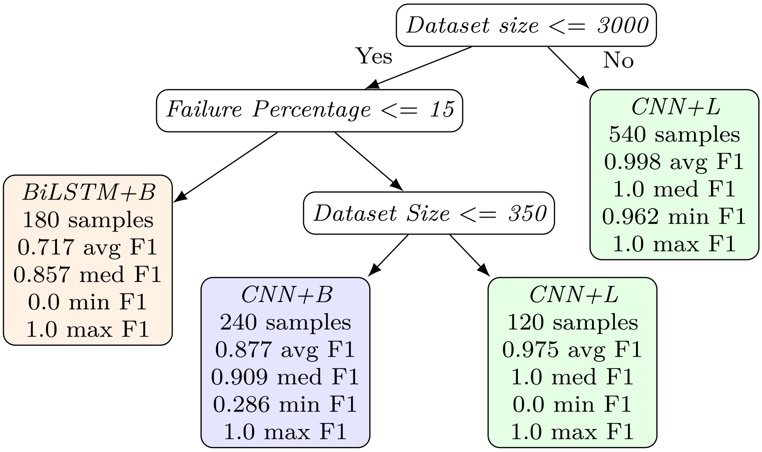


Fig. 13 Decision Tree identifying the best configurations based on dataset characteristics

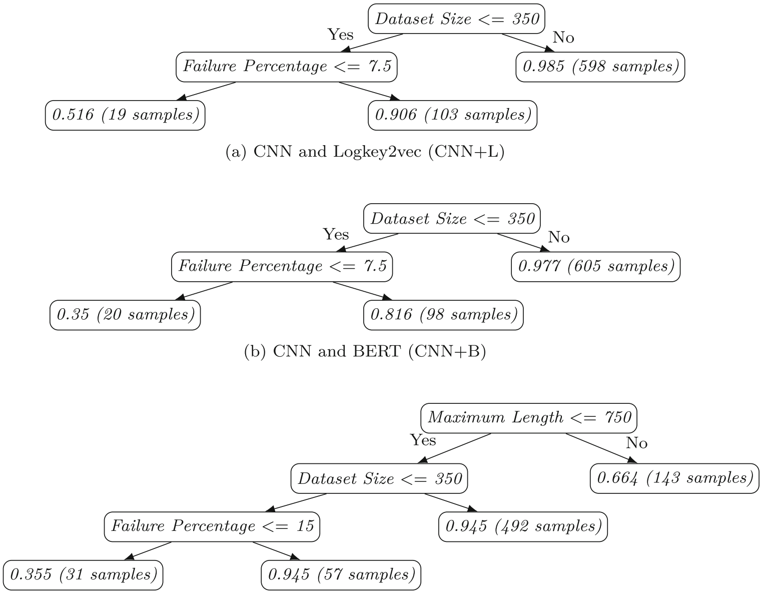
and CNN+B. We suspect this result is due to BiLSTM+B's higher capability in the presence of highly imbalanced datasets. In contrast, if the failure percentage is above 15%, CNN+B is recommended when the dataset size is lower than or equal to 350, while CNN+L is recommended when the dataset size is higher than 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, leading 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 14 depicts the regression trees for CNN+L (Fig. 14a), CNN+B (Fig. 14b), and BiSLTM+B (Fig. 14c). For example, in Fig. 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 Sect. 7.1.



(a) CNN and Logkey2vec (CNN+L)

(b) CNN and BERT (CNN+B)

(c) BiLSTM and BERT (BiLSTM+B)

Fig. 14 Regression Tree for the best configurations based on dataset characteristics in F1 scores

Table 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
Synthesised$_{\mathcal{M}_1}$	1000	500	20	0.987	1.000	0.993
Synthesised$_{\mathcal{M}_2}$	1000	500	20	0.932	0.965	0.9480
Synthesised$_{\mathcal{M}_3}$	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

> The answer to RQ4 is that dataset size, followed by failure percentage, plays an important role in the accuracy of DL-based failure predictors while LSL is important only for some 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 dataset size is above 350 or failure percentage is above 7.5%.

6.5 RQ5: Real-world Data

Table 9 shows the accuracy results of synthesized datasets alongside those of the real-world dataset, OpenStack_FP, for the best the same DL-based configuration, CNN+L. The “Dataset” column lists the datasets chosen for comparison. Synthesised$_{\mathcal{M}_1}$, Synthesised$_{\mathcal{M}_2}$, and Synthesised$_{\mathcal{M}_3}$ 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, there are two dataset instances matching these three characteristics but having 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_PF, the table presents an average of the two synthesised datasets in each row dedicated to Synthesised data. Furthermore, the fourth row shows the average accuracy results from all synthesised behavioural models.

According to Table 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 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_PF; for each test, the accuracy results from the synthesized datasets were paired with the results from OpenStack_PF. All p-values for precision, recall, and F1 score are far above 0.05, indicating that the differences between real-world and synthesized 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).

6.6 Data Availability Statement

The replication package, including the implementation, generated datasets with behavioural models, and results, is publicly available (Hadadi et al. 2024).

7 Discussion

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 1, the full configuration of DL encoders and LSE strategies are evaluated. Moreover, instead of using a limited number of datasets, using synthesized data enables us to control dataset characteristics to identify necessary conditions for achieving high-accuracy models. Nonetheless, we also considered a real-world dataset for failure prediction (OpenStack_PF) and applied it to the best failure predictor configuration. This allows us to compare the failure prediction accuracy results obtained on the synthesized datasets with those obtained on the OpenStack FP dataset.

Several major findings are reported in § 6. First, the CNN-based DL encoder fares the best among different DL encoders, including the ones 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 increasing the maximum length of log sequences significantly decreases the failure prediction accuracy.

Fifth, based on the analysis of § 6.4, we are able to 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 if the dataset size is 350 or below, and the failure percentage is 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 when the dataset size is 350 or below, similar to CNN+L and CNN+B, and the failure percentage is 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 might not be easy to control in practice, collecting more log sequences during the operation of the system to increase the dataset size is usually feasible.

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 (Notaro et al. 2021). 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 § 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 § 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 § 6.6), enabling AIOps engineers to reuse our artifacts seamlessly.

Software Engineering Researchers In this paper, 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 (Das et al. 2018, 2020), we propose a novel approach to study configurations of LSE strategies and DL architectures that have not been studied together before (see Table 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 § 5.3 can be adopted to generate synthetic datasets tailored to specific requirements.

7.2 Threats to Validity

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

Hyper-parameter Tuning of Models The hyper-parameters 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 § 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 § 5.3.1, we proposed a new approach, with precise algorithms, that can generate datasets in a controlled, unbiased manner as discussed in § 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 § 5.3.6. To further support the validity of the generation process, in § 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 to know 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 the generation of synthetic datasets may have a significant impact on the experimental results. We want to remark 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 § 5.4 and provided a remark on its generalizability in § 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, the log sequence embedding strategies, the dataset generation algorithms, and the scripts used in our experiments might include unexpected bugs. To mitigate this risk, we used the replication packages of existing studies (Chen et al. 2021; Le and Zhang 2021) as much as possible. Also, we carefully performed code reviews.

8 Conclusion

In this paper, 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 which 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 three behavioural models.

Evaluation results show that the accuracy of the CNN-based encoder is significantly higher than 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 failure percentage increases 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 synthesized 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 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 noises 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 § 7.2, when using real-world data, we also plan to include additional evaluation metrics, such as lead time (Salfner et al. 2010) 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 1621 joint research program grant, the 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 1623 by Digital Alliance of Canada (alliancecan.ca).

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Basin D, Dardinier T, Heimes L, Krstić S, Raszyk M, Schneider J, Traytel D (2020) 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, Springer-Verlag, Berlin, Heidelberg, pp 432–453, https://doi.org/10.1007/978-3-030-51074-9_25

Bauer E, Adams R (2012) Reliability and availability of cloud computing. John Wiley & Sons

Black PE (2020) Strongly connected component. Dictionary of Algorithms and Data Structures https://www.nist.gov/dads/HTML/stronglyConnectedCompo.html

Blom J, Hessel A, Jonsson B, Pettersson P (2005) Specifying and generating test cases using observer automata. Lecture Notes Comput Sci 3395:125–139. https://doi.org/10.1007/978-3-540-31848-4_9

Bogatinovski J, Nedelkoski S, Wu L, Cardoso J, Kao O (2022) Failure identification from unstable log data using deep learning. In: 2022 22nd IEEE International symposium on cluster, cloud and internet computing (CCGrid) pp 346–355, https://api.semanticscholar.org/CorpusID:247996709

Bombarda A, Gargantini A (2020) An Automata-Based Generation Method for Combinatorial Sequence Testing of Finite State Machines. In: Proceedings - 2020 IEEE 13th International conference on software testing, verification and validation workshops, ICSTW 2020 pp 157–166. https://doi.org/10.1109/ICSTW50294.2020.00036

Breiman L (2001) Random forests. Mach Learn 45(1):5–32. https://doi.org/10.1023/A:1010933404324

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and Regression Trees. Wadsworth

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984b) Classification and Regression Trees. Chapman and Hall/CRC

Carvalho TP, Soares FAAMN, Vita R, da P Francisco R, Basto JP, Alcalá SGS, (2019) A systematic literature review of machine learning methods applied to predictive maintenance. Comput & Industrial Eng 137:106024. https://doi.org/10.1016/j.cie.2019.106024, https://www.sciencedirect.com/science/article/pii/S0360835219304838

Chen Y, Yang X, Lin Q, Zhang D, Dong H, Xu Y, Li H, Kang Y, Zhang H, Gao F, Xu Z, Dang Y (2019) Outage prediction and diagnosis for cloud service systems. In: The Web Conference 2019 - Proceedings of the World Wide Web Conference, WWW 2019 pp 2659–2665. https://doi.org/10.1145/3308558.3313501

Chen Y, Li L, Li W, Guo Q, Du Z, Xu Z (2022) AI Computing Systems: An Application Driven Perspective. Elsevier Sci https://books.google.ca/books?id=RSWJEAAAQBAJ

Chen Z, Liu J, Gu W, Su Y, Lyu MR (2021) Experience report: Deep learning-based system log analysis for anomaly detection. https://doi.org/10.48550/ARXIV.2107.05908, https://arxiv.org/abs/2107.05908

Cho K, Van Merriënboer B, Bahdanau D, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv:1406.1078

Chollet F (2017) Xception: Deep learning with depthwise separable convolutions. In: IEEE Conference on computer vision and pattern recognition (CVPR)

Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297

Cotroneo D, De Simone L, Liguori P, Natella R, Bidokhti N (2019) How bad can a bug get? An empirical analysis of software failures in the OpenStack cloud computing platform. In: 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, https://doi.org/10.1145/3338906.3338916, arXiv:1907.04055

Das A, Mueller F, Siegel C, Vishnu A (2018) Desh: Deep learning for system health prediction of lead times to failure in HPC. In: HPDC 2018 - Proceedings of the 2018 International symposium on high-performance parallel and distributed computing pp 40–51, https://doi.org/10.1145/3208040.3208051

Das A, Mueller F, Rountree B (2020) Aarohi: Making Real-Time Node Failure Prediction Feasible. In: Proceedings - 2020 IEEE 34th International parallel and distributed processing symposium, IPDPS 2020 pp 1092–1101, https://doi.org/10.1109/IPDPS47924.2020.00115

Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805

Digital Research Alliance of Canada (2016) https://alliancecan.ca/, accessed: March 2, 2023

Ding M, Zhou C, Yang H, Tang J (2020) Cogltx: Applying bert to long texts. In: Neural information processing systems

Du M, Li F, Zheng G, Srikumar V (2017) Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '17, p 1285–1298, https://doi.org/10.1145/3133956.3134015, Dwivedi VP, Luu AT, Laurent T, Bengio Y, Bresson X (2021) Graph neural networks with learnable structural and positional representations. arXiv:2110.07875

Fernández-Delgado M, Cernadas E, Barro S, Amorim D (2014) Do we need hundreds of classifiers to solve real world classification problems? J Mach Learn Res 15:3133–3181

Foundation CC (2023) Common crawl corpus. https://commoncrawl.org/

Gers FA, Schmidhuber JA, Cummins FA (2000) Learning to forget: Continual prediction with lstm. Neural Comput 12(10):2451–2471. https://doi.org/10.1162/089976600300015015

Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, Liu T, Wang X, Wang G, Cai J, Chen T (2018) Recent advances in convolutional neural networks. Pattern Recognition 77:354–377, https://doi.org/10.1016/j.patcog.2017.10.013, https://www.sciencedirect.com/science/article/pii/S0031320317304120

Guo H, Yuan S, Wu X (2021) Logbert: Log anomaly detection via bert. In: 2021 International joint conference on neural networks (IJCNN), pp 1–8, https://doi.org/10.1109/IJCNN52387.2021.9534113

Hadadi F, Dawes J, Shin D, Bianculli D, Briand L (2024) Replication package. https://doi.org/10.6084/m9.figshare.22219111, https://figshare.com/articles/software/Replication_Package/22219111

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21:1263–1284, https://api.semanticscholar.org/CorpusID:206742563

He S, He P, Chen Z, Yang T, Su Y, Lyu MR (2021) A Survey on Automated Log Analysis for Reliability Engineering. ACM Comput Surv 54(6), https://doi.org/10.1145/3460345, arXiv:2009.07237

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Huang S, Liu Y, Fung C, He R, Zhao Y, Yang H, Luan Z (2020) HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log. IEEE Trans Netw Service Manag 17(4):2064–2076. https://doi.org/10.1109/TNSM.2020.3034647

Huang Z, Xu W, Yu K (2015) Bidirectional lstm-crf models for sequence tagging. https://doi.org/10.48550/ARXIV.1508.01991, arXiv:1508.01991

Johnson JM, Khoshgoftaar TM (2019) Survey on deep learning with class imbalance. J Big Data 6(1), https://doi.org/10.1186/s40537-019-0192-5

Joulin A, Grave E, Bojanowski P, Douze M, Jégou H, Mikolov T (2016) Fasttext.zip: Compressing text classification models. arXiv:1612.03651

Kim Y (2014) Convolutional neural networks for sentence classification. arXiv:1408.5882

Kingma DP, Ba J (2015) Adam: A method for stochastic optimization. In: Bengio Y, LeCun Y (eds) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, arXiv:1412.6980

Kluge F, Rochange C, Ungerer T (2017) EMSBench: Benchmark and Testbed for Reactive Real-Time Systems. Leibniz Trans Embedded Syst 4(2):02–1–02:23, https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v004-i002-a002

Krstić S, Schneider J (2020) A Benchmark Generator for Online First-Order Monitoring, vol 12399. Springer International Publishing, LNCS. https://doi.org/10.1007/978-3-030-60508-7_27

Le VH, Zhang H (2021) Log-based anomaly detection without log parsing. In: 2021 36th IEEE/ACM International conference on automated software engineering (ASE), pp 492–504, https://doi.org/10.1109/ASE51524.2021.9678773

Le VH, Zhang H (2022) Log-based anomaly detection with deep learning: How far are we? In: Proceedings of the 44th international conference on software engineering, association for computing machinery, New York, NY, USA, ICSE ’22, p 1356–1367, https://doi.org/10.1145/3510003.3510155

Li X, Chen P, Jing L, He Z, Yu G (2020) Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In: 2020 IEEE 31st International symposium on software reliability engineering (ISSRE), IEEE computer society, Los Alamitos, CA, USA, pp 92–103, https://doi.org/10.1109/ISSRE5003.2020.00018, https://doi.ieeecomputersociety.org/10.1109/ISSRE5003.2020.00018

Lin Q, Hsieh K, Dang Y, Zhang H, Sui K, Xu Y, Lou JG, Li C, Wu Y, Yao R, Chintalapati M, Zhang D (2018) 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, association for computing machinery, New York, NY, USA, ESEC/FSE 2018, p 480–490, https://doi.org/10.1145/3236024.3236060

Lipton ZC (2015) A critical review of recurrent neural networks for sequence learning. arXiv:1506.00019

Liu X, He Y, Liu H, Zhang J, Liu B, Peng X, Xu J, Zhang J, Zhou A, Sun P, Zhu K, Nishi A, Zhu D, Zhang K (2020) Smart Server Crash Prediction in Cloud Service Data Center. In: 2020 19th IEEE Intersociety conference on thermal and thermomechanical phenomena in electronic systems (ITherm), https://doi.org/10.1109/ITherm45881.2020.9190321

Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V (2019) 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)

Lu S, Wei X, Li Y, Wang L (2018) Detecting anomaly in big data system logs using convolutional neural network. IEEE Access 6:21929–21940. https://doi.org/10.1109/ACCESS.2018.2811530

Matsumoto M, Nishimura T (1998) Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Trans Model Comput Simulation 8(1):3–30. https://doi.org/10.1145/272991.272995

Meng W, Liu Y, Zhu Y, Zhang S, Pei D, Liu Y, Chen Y, Zhang R, Tao S, Sun P, Zhou R (2019) Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In: International joint conference on artificial intelligence

Meng W, Liu Y, Huang Y, Zhang S, Zaiter F, Chen B, Pei D (2020) A semantic-aware representation framework for online log analysis. In: 2020 29th International conference on computer communications and networks (ICCCN), pp 1–7, https://doi.org/10.1109/ICCCN49398.2020.9209707

Mikolov T, Chen K, Corrado GS, Dean J (2013) Efficient estimation of word representations in vector space. In: International conference on learning representations

Nedelkoski S, Bogatinovski J, Acker A, Cardoso J, Kao O (2020) Self-attentive classification-based anomaly detection in unstructured logs. Proceedings - IEEE international conference on data mining, ICDM 2020-Novem(Icdm):1196–1201, https://doi.org/10.1109/ICDM50108.2020.00148, arXiv:2008.09340

Notaro P, Cardoso J, Gerndt M (2021) A survey of aiops methods for failure management. ACM Trans Intell Syst Technol 12(6), https://doi.org/10.1145/3483424

O'Shea K, Nash R (2015) An introduction to convolutional neural networks. https://doi.org/10.48550/ARXIV.1511.08458, arXiv:1511.08458

Package RP (2019) https://docs.python.org/3/library/random.html, accessed 2022-11-14

Prechelt L (1998) Early stopping-but when? In: Neural Networks: Tricks of the Trade, Springer, pp 55–69

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language models are unsupervised multitask learners. In: Neural information processing systems

Rajaraman A, Leskovec J, Ullman J (2014) Mining of Massive Datasets. Cambridge University Press. https://doi.org/10.1017/CBO9781139058452

Russo B, Succi G, Pedrycz W (2015) Mining system logs to learn error predictors: a case study of a telemetry system. Empirical Softw Eng 20(4):879–927. https://doi.org/10.1007/s10664-014-9303-2

Sahoo RK, Oliner AJ, Rish I, Gupta M, Moreira JE, Ma S, Vilalta R, Sivasubramaniam A (2003) Critical event prediction for proactive management in large-scale computer clusters. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining pp 426–435, https://doi.org/10.1145/956750.956799

Salfner F, Lenk M, Malek M (2010) A survey of online failure prediction methods. ACM Comput Surveys 42(3), https://doi.org/10.1145/1670679.1670680

Schuster M, Paliwal K (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681. https://doi.org/10.1109/78.650093

Shin D, Bianculli D, Briand L (2022) Prins: Scalable model inference for component-based system logs. Empirical Softw Engg 27(4), https://doi.org/10.1007/s10664-021-10111-4

Sun C, Qiu X, Xu Y, Huang X (2019) How to fine-tune bert for text classification? In: Sun M, Huang X, Ji H, Liu Z, Liu Y (eds) Chinese Computational Linguistics. Springer International Publishing, Cham, pp 194–206

Tauber A (2018) exrex: Irregular methods for regular expressions. https://github.com/asciimoo/exrex, accessed 2022-11-14

Upton G, Cook I (2008) A Dictionary of Statistics. Oxford Paperback Reference, OUP Oxford, https://books.google.ca/books?id=u97pzxRjaCQC

Vaswani A, Shazeer NM, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. arXiv:1706.03762

Walkinshaw N, Taylor R, Derrick J (2013) Inferring extended finite state machine models from software executions. In: 2013 20th Working conference on reverse engineering (WCRE), pp 301–310, https://doi.org/10.1109/WCRE.2013.6671305

Weijie D, Yunyi L, Jing Z, Xuchen S (2021) Long text classification based on bert. In: 2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC), vol 5, pp 1147–1151, https://doi.org/10.1109/ITNEC52019.2021.9587007

Wu X, Li H, Khomh F (2023) On the effectiveness of log representation for log-based anomaly detection. arXiv:2308.08736

Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, Klingner J, Shah A, Johnson M, Liu X, Kaiser L, Gouws S, Kato Y, Kudo T, Kazawa H, Stevens K, Kurian G, Patil N, Wang W, Young C, Smith J, Riesa J, Rudnick A, Vinyals O, Corrado G, Hughes M, Dean J (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:1609.08144

Xie Y, Zhang H, Babar MA (2022) Loggd: Detecting anomalies from system logs with graph neural networks. In: 2022 IEEE 22nd International conference on software quality, reliability and security (QRS), pp 299–310, https://doi.org/10.1109/QRS57517.2022.00039

Xu P, Kumar D, Yang W, Zi W, Tang K, Huang C, Cheung JCK, Prince S, Cao Y (2020) Optimizing deeper transformers on small datasets. In: Annual meeting of the association for computational linguistics

Yamanishi K, Maruyama Y (2005) Dynamic syslog mining for network failure monitoring. In: Proceedings of the Eleventh ACM SIGKDD International conference on knowledge discovery in data mining, association for computing machinery, New York, NY, USA, KDD '05, p 499–508, https://doi.org/10.1145/1081870.1081927

Yang L, Chen J, Wang Z, Wang W, Jiang J, Dong X, Zhang W (2021) Semi-supervised log-based anomaly detection via probabilistic label estimation. In: 2021 IEEE/ACM 43rd International conference on software engineering (ICSE), pp 1448–1460, https://doi.org/10.1109/ICSE43902.2021.00130

Zhang S, Liu Y, Meng W, Luo Z, Bu J, Yang S, Liang P, Pei D, Xu J, Zhang Y, Chen Y, Dong H, Qu X, Song L (2018) Prefix: Switch failure prediction in datacenter networks. Proc ACM Meas Anal Comput Syst 2(1):2:1–2:29, https://doi.org/10.1145/3179405

Zhang X, Xu Y, Lin Q, Qiao B, Zhang H, Dang Y, Xie C, Yang X, Cheng Q, Li Z, Chen J, He X, Yao R, Lou JG, Chintalapati M, Shen F, Zhang D (2019) Robust log-based anomaly detection on unstable log data. In: ESEC/FSE 2019 - Proceedings of the 2019 27th ACM joint meeting european software engineering conference and symposium on the foundations of software engineering pp 807–817, https://doi.org/10.1145/3338906.3338931

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Fatemeh Hadadi is currently working toward a Ph.D. degree under the supervision of Prof. Lionel Briand with the School of EECS, University of Ottawa, and a member of Nanda Lab. In 2021, she was awarded the prize of first class in computer science from the supporter foundation of the University of Tehran and was granted a direct Ph.D. from B.C. by the University of Ottawa. Her research interests include log analysis (e.g., failure prediction and anomaly detection), natural language processing, and applied machine learning.

Joshua H. Dawes completed his PhD at CERN (Geneva, Switzerland) and was awarded the final degree by the University of Manchester (Manchester, UK) in 2021. During his time at CERN, Josh developed and applied formal verification techniques for software that served operations of the Large Hadron Collider. After CERN he spent time at SnT in Luxembourg where, while he was helping with this paper, his formal verification software was applied by companies in the Avionics and Acoustics sectors. Having concluded his stint in academia, Josh is now a Senior Software Engineering Manager at the ISIS Neutron and Muon Source in the UK.

Donghwan Shin did his BS, MSc, and PhD at Korea Advanced Institute of Science and Technology (KAIST), South Korea. This was followed by four years as a research associate/scientist at the SVV (Software Verification and Validation) group, the Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT) of the University of Luxembourg. He is currently a lecturer (assistant professor in the American system) at the Department of Computer Science, University of Sheffield, UK. His research and teaching interests lie in testing for ML-enabled cyber-physical systems (e.g., ML-enabled automated driving systems), log analysis (e.g., model inference and anomaly detection), and mutation testing. More details can be found at https://dshin.info.

Domenico Bianculli is associate professor/chief scientist 2 at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He holds a PhD degree from Università della Svizzera italiana (Lugano, Switzerland), a MSc in Computing Systems Engineering and a BSc in Computer Engineering, both from Politecnico di Milano (Milan, Italy). Domenico's research focuses on the specification and verification of evolvable software systems. His research interests include: run-time verification, temporal logics and specification languages, log analysis, program analysis, and regulatory compliance.

Lionel C. Briand is professor of software engineering and has shared appointments between (1) The University of Ottawa, Canada, and (2) The Lero SFI Centre—the national Irish centre for software research—hosted by the University of Limerick, Ireland. In collaboration with colleagues, for over 30 years, he has run many collaborative research projects with companies in the automotive, satellite, aerospace, energy, financial, and legal domains. Lionel has held various engineering, academic, and leading positions in seven countries. He currently holds a Canada Research Chair (Tier 1) on "Intelligent Software Dependability and Compliance" and is the director of Lero, the national Irish re for software research. Lionel was elevated to the grades of IEEE Fellow and ACM Fellow for his work on software testing and verification. Further, he was granted the IEEE Computer Society Harlan Mills award, the ACM SIGSOFT outstanding research award, and the IEEE Reliability Society engineer-of-the-year award. He also received an ERC Advanced grant in 2016 on modelling and testing cyber-physical systems, the most prestigious individual research award in the European Union and was elected a fellow of the Academy of Science, Royal Society of Canada in 2023. More details can be found at: http://www.lbriand.info.

Authors and Affiliations

Fatemeh Hadadi[1] · Joshua H. Dawes[4] · Donghwan Shin[5] · Domenico Bianculli[4] · Lionel Briand[1,2,3]

✉ Fatemeh Hadadi
fhada072@uottawa.ca

Joshua H. Dawes
joshua.dawe@uni.lu

Donghwan Shin
d.shin@sheffield.ac.uk

Domenico Bianculli
domenico.bianculli@uni.lu

Lionel Briand
lbriand@uottawa.ca

[1] University of Ottawa, Ottawa, Canada

[2] Lero SFI Centre for Software Research, Limerick, Ireland

[3] University of Limerick, Limerick, Ireland

[4] SnT Centre, University of Luxembourg, Esch-sur-Alzette, Luxembourg

[5] University of Sheffield, United Kingdom (Part of this work was done while the author was with the University of Luxembourg), Sheffield, Luxembourg