Virtual Machine Proactive Fault Tolerance using Log-based Anomaly Detection

Pratheek Senevirathne
Index number: 19001622

Supervisor: Dr. Dinuni Fernando
Co-Supervisor: Dr. Jerome Dinal Herath

April 2024

Submitted in partial fulfillment of the requirements of the
B.Sc in Computer Science Final Year Project (SCS4224)

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Student Name : P. L. W. Senevirathne
Registration Number : 2019/CS/162
Index Number : 19001622

Signature & Date

This is to certify that this dissertation is based on the work of Mr. P. L. W. Senevirathne under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name : Dr. Dinuni K. Fernando

Signature & Date

Co-supervisor Name : Dr. Jerome Dinal Herath

Signature & Date

Abstract

Virtual Machine (VM) fault tolerance ensures high availability in cloud computing environments. Proactive fault tolerance strategies, which identify potential failures before they occur and move VMs to healthy hosts, help avert service disruptions due to VM failures. In recent years, there has been a wide adoption of Machine Learning (ML) approaches for fault detection. However, existing approaches often rely on ML models trained on labeled data, which can be challenging to obtain. They also require a large amount of training data and may struggle with real-time failure prediction and fast adaptation to dynamic environments. In this work, we propose VMFT-LAD (Virtual Machine Proactive Fault Tolerance using Log-based Anomaly Detection), a semi-supervised log anomaly detection model for proactive VM fault tolerance. VMFT-LAD leverages the efficiency of the Matrix Profile for anomaly detection and the log inference capability of Large Language Models (LLMs) to continuously learn and identify potential failures, including unforeseen fault types, with minimal human intervention. By focusing on detecting anomalies in log data, our approach operates without the need for labeled failure data. Extensive evaluations on several datasets demonstrate VMFT-LAD's outstanding performance, achieving a Numenta Anomaly Benchmark (NAB) standard score of 90.74 under the criterion of predicting failures before the failure point. Compared to state-of-the-art anomaly detection models, VMFT-LAD demonstrates a high early detection rate of 96.28% while maintaining a low false positive rate of 0.02%, enabling timely VM migration before failures occur. The results highlight the superiority of VMFT-LAD in facilitating reliable and proactive fault tolerance strategies in virtualized environments.

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my supervisors, Dr. Dinuni Fernando and Dr. Dinal Herath, for their invaluable guidance, unwavering support, continuous encouragement, and motivation throughout this research project. Their insightful feedback and expertise were invaluable in shaping the direction of this project and ensuring its successful completion.

I would like to thank Samindu Cooray, who assisted me in setting up the servers at the Network Operations Center (NOC) at UCSC and provided the necessary migration time data needed for the evaluation of this work. I should also thank WSO2 for providing the servers and Dr. Dinuni Fernando for facilitating their collaboration, without which this project would not have been possible.

I would like to extend my appreciation to the staff at NOC for their hard work in setting up the servers and granting us access to the server room. I should also thank all the servers I used for this project, for not failing up on me, even though I stressed them to their limits.

Finally, I am eternally grateful to my parents, siblings, and colleagues for their unwavering support, encouragement, and understanding throughout this challenging journey. Their presence and belief in me were a constant source of motivation throughout the project.

Contents

List of Acronyms

VM Virtual Machine

OS Operating System

VMM Virtual Machine Monitor

PM Physical Machine

AWS Amazon Web Services

GCP Google Cloud Platform

CDC Cloud Data Center

FT Fault Tolerance

SLA Service Level Agreement

ML Machine Learning

CI Continuous Integration

CD Continuous Delivery

CNN Convolutional Neural Network

LSTM Long Short-Term Memory

FNN Feed-forward Neural Network

SVM Support Vector Machine

VNF Virtual Network Function

NFV Network Functions Virtualization

vEPC Virtual Evolved Packet Core

IDS Intrusion Detection System

NLP Natural Language Processing

NLU Natural Language Understanding

BERT Bidirectional Encoder Representations from Transformers

NAB Numenta Anomaly Benchmark

1 Introduction

A Virtual Machine (VM) can be thought of as a software emulation of a physical computer. This technology allows several Operating Systems (OSs) to run on a single computer/server, each with its own virtualized hardware, and each VM functions separately from the other VMs in the same Physical Machine (PM). VMs are used for a variety of purposes, including testing and development, application deployment, building Continuous Integration (CI)/Continuous Delivery (CD) pipelines, server consolidation, running legacy applications, etc. They are managed by the hypervisor, which is also known as a Virtual Machine Monitor (VMM), a piece of software that creates and manages VMs.

As with any system, VMs are failure-prone. VM failures may occur at any moment of its execution or during migration from one host (PM) to another. VM failure can be seen as a failure in the end-user application/service running in the VM. An end-user application/service may experience a failure when there is a failure or an error in the PM (host) hardware components, network, or in software components such as the host OS, hypervisor, VM instance, the guest OS running in the VM, or due to a failure in the end-user application itself [1].

VM failure can cause downtime and loss of user data, and disrupt service availability, leading to negative impacts on users and business operations. By utilizing failure prediction and proactive VM migration techniques, we can greatly reduce service downtime due to VM failures. One effective approach for proactive VM migration is live migration, which enables VMs to be migrated to a healthy PM while maintaining service continuity [2], [3].

All of the above-mentioned software components related to VMs generate logs that contain valuable information about systems' states and events, including fault and failure data. Using Machine Learning (ML) techniques to analyze these logs, it is possible to predict VM failures and proactively move VMs from faulty PMs to healthy ones before the failure occurs [4].

1.1 Background

1.1.1 Virtual Machine Live Migration

The live migration technique tries to migrate the VM while it is switched on, and the applications are still running in them with minimal interruptions to the running applications [5]. The main objectives of this approach are to optimize the end-user application performance and to minimize downtime. There are three

sub-categories: Pre-copy migration, Post-copy migration, and hybrid approach. Figure 1 illustrates the timeline of pre-copy vs post-copy methods.

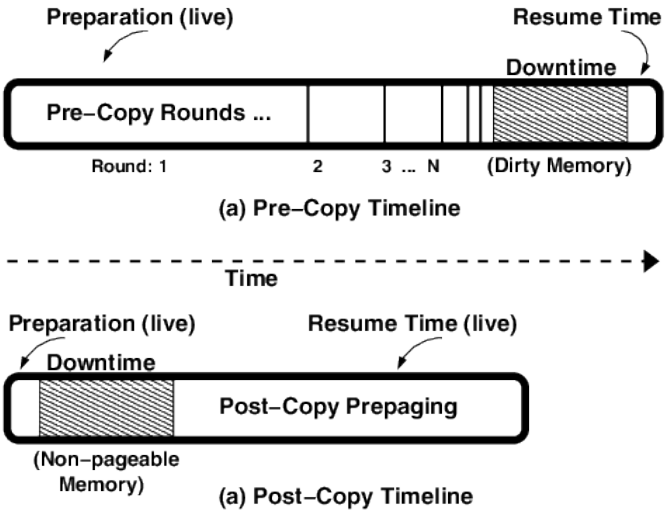


Figure 1: Pre-copy vs Post-copy timeline [6]

1. **Pre-copy:**

   A pre-copy migration scheme will migrate the VM from a probable failing node to a target node by iteratively copying all of its memory content before stopping it in the source node and activating it in the target node. This migration scheme is resource intensive, and the total migration time is also comparatively high [7].

2. **Post-copy:**

   This scheme will migrate the VM from the host node to the target node by immediately suspending the VM and capturing the minimum possible state to migrate the VM. This is then moved to the target, and the VM is started in the target. And if the guest OS requires a page to read/write that is not already copied, the migration system will copy it to the VM memory on the fly over the network. When all the pages are copied from the source server, the connection to the server will be terminated, and the source VM can then be terminated [8].

3. **Hybrid approach:**

   These frameworks integrate the properties of both the previous methods to reduce migration time and improve system performance. It includes a finite

pre-copy stage before moving on to migrating the VM and running the post-copy stage. This greatly reduces the page faults that may occur in the future as a large amount of the memory is already copied, so this method reduces the workload on the network and improves the application performance[9].

All the existing VM migration schemes can be classified mainly into two groups, namely, proactive migration and reactive migration [3], [9].

Reactive Virtual Machine Migration: This method is the most widely adopted and used method in cloud VM Fault Tolerance (FT). Reactive migration deals with faults after they have happened; that is, they migrate the VM to a suitable destination node after a fault/failure has been detected. Using this method will take some time to get the failed VM properly running back again, which may lead to Service Level Agreement (SLA) violation. Even though this method is undesirable, one advantage of this method is that the overhead associated with running a machine learning algorithm to predict fault occurrence is completely avoided in this mechanism [9].

Proactive Virtual Machine Migration: These frameworks continuously monitor the system and predict fault occurrence using an ML or a similar approach, and if a VM or the PM is predicted to fail, it will migrate the VM(s) to a chosen healthy PM. This work is focused on proactive migration, where the failure is predicted using an ML approach by analyzing logs.

1.1.2 Logs, Log Keys and Values

Logs are files that contain information about events that occur in a computer system. These events can happen in the OS, in applications, or hardware devices. Logs contain system events, such as startup and shutdown; hardware changes; application events, such as errors, warnings, and performance metrics; security events, such as login attempts, failed access attempts, etc.

The events are logged in the log file as log lines, and the structure of each log file is different. Usually, each log line contains the timestamp at which it was logged, the log verbosity level (INFO, WARNING, SEVERE, etc.), the application information, and the actual log data. The log messages are text printed by logging statements in program code such as *print(), logging.info(), logger.log()* written by the program developers [10].

Following is a small extract from the kernel log file,

```
Sep 12 09:54:45 cloudnet2 kernel: [93248.482863] br0: port 2(tap1)
   entered disabled state
```

```
Sep 12 09:54:45 cloudnet2 kernel: [93248.482991] br0: port 4(tap3)
    entered disabled state
Sep 12 12:10:46 cloudnet2 kernel: [ 8132.463411] FS-Cache: Loaded
Sep 12 12:10:46 cloudnet2 kernel: [ 8132.541471] FS-Cache: Netfs 'nfs
    ' registered for caching
```

The unstructured log data needs to be parsed in some way to make them structured, efficient, and easier to analyze [11]. In this study, we utilize the log parsing technique, which transforms the log messages into log keys and values. For this, the parser must distinguish the constant and variable parts of each raw log message. The parser assigns a unique identifier for each constant part of the log line; this identifier is the log key. The variable part is extracted as the log parameters [10], [12].

The constant part is analogous to the log printing statement in the program source code [11]; for example,

```
printf("%s: port %d(%s) entered %s state", bridgeName, portId,
    portName, portState)
```

The parser outputs parsed log lines as log key and value pairs. Table 1 presents the sample parsed log data output for the above log lines. These values can then be directly used in log analysis.

Parsed log line	Log key	Log values
br0: port ⟨*⟩(tap⟨*⟩) entered ⟨*⟩ state	101	[2, 1, disabled]
br0: port ⟨*⟩(tap⟨*⟩) entered ⟨*⟩ state	101	[4, 3, disabled]
FS-Cache: Loaded	102	[]
FS-Cache: Netfs ⟨*⟩ registered for caching	103	[nfs]

Table 1: Sample Parsed log lines

Most of the logs will not have any failure or failure indicating logs because the VM will not fail under regular operation. Waiting for the VMs to fail is not feasible because a VM may not fail in its usual operation and may take months or years to fail under normal conditions. To tackle this issue, we need a way to simulate VM and host failures; fault injection is an effective way of simulating the VM and host failures [4].

1.1.3 Fault Injection

Fault injection is a software testing technique used to intentionally introduce faults or errors into a system or component to evaluate its behavior and robustness in

the presence of unexpected or adverse conditions. The primary purpose of fault injection is to assess how a system responds to faults, errors, or failures and to identify weaknesses in its design or implementation. In this context, we use fault injection software to simulate host and VM failure scenarios. Simulating high resource demands and hardware faults allows for identifying and collecting near-failure and failure logs. The near-failure logs will contain warning messages, fault logs, and other valuable logs for VM failure prediction.

1.1.4 Machine Learning for Log Analysis

We can utilize several supervised, semi-supervised, and unsupervised ML techniques for log analysis. Recent studies on the subject [4], [13], [14], have primarily utilized supervised ML models for failure prediction by analyzing the logs, and they have shown that several ML models prove to be effective, and the authors have shown that Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) models have shown superior performance over other models in the reviewed papers. However, supervised models are not preferred because they may fail to identify failure situations that they are not trained on.

Some ML models, like CNN, require the logs to be in a numerical format, such as vectors. There are several approaches to achieve this conversion. One common method is to employ word embedding, which involves using a language model to convert log words into tokens and then represent them as vectors. This technique captures the semantic meaning of the words and their relationships within the log data [4]. Alternatively, we can also consider converting the logs to a log event count representation, where each log event is treated as a feature, and the count of occurrences of each event is recorded [14].

In the Literature Review section, we will discuss how the authors of the related papers have approached this conversion step and which ML algorithms they have used, providing further insights into their methodologies.

This study mainly focuses on developing a semi-supervised model using the above-mentioned log key and value representation for this conversion step because of its simplicity and speed.

1.1.5 Log Key Subsequences

To extract log key subsequences, we first obtain the timestamp and log message from the log files. Using an online log parser like Drain [15], we parse the log message to extract the log template and its corresponding identifier (log key).

The output, consisting of the timestamp and the log key, forms a time series of log keys. Given a log key time series $T = t_1, t_2, \ldots, t_{n-1}, t_n$, the subsequence of length m at a fixed position i is defined as $T_{i,m} = t_{i-m+1}, t_{i-m+2}, \ldots, t_{i-1}, t_i$, where t_i is the i^{th} log key of the time series T, and $i \geq m, i \leq n$.

The distance between any two such subsequences can be calculated using various vector distance measures. The most commonly used measure, employed in the Matrix Profile [16], is the z-normalized Euclidean distance. For two given subsequences $T_{i,m}$ and $T_{j,m}$, the z-normalized Euclidean distance is defined as:

$$D^E(T_{i,m}, T_{j,m}) = \left\| \frac{T_{i,m} - \mu_i}{\sigma_i} - \frac{T_{j,m} - \mu_j}{\sigma_j} \right\|_2, \tag{1}$$

where μ_i, μ_j and σ_i, σ_j are the mean and standard deviation of the subsequences $T_{i,m}$ and $T_{j,m}$, respectively.

For a given subsequence $T_{i,m}$, the vector of distances between $T_{i,m}$ and each subsequence of T is called the distance profile, denoted as $D_i = d_{i,m}, d_{i,m+1}, \ldots, d_{i,n-1}, d_{i,n}$, where $d_{i,j}$ is the z-normalized Euclidean distance between $T_{i,m}$ and $T_{j,m}$.

Using the distance profile for a subsequence $T_{i,m}$, we can quickly identify the closest matching subsequence to it, excluding its trivial match, by finding the minimum distance profile value. The matrix profile P is defined as the vector that stores this minimum distance for each subsequence in the time series T, i.e., $P = min(D_m), min(D_{m+1}), \ldots, min(D_{n-1}), min(D_n)$. A small value in the Matrix Profile suggests that the subsequence pattern appears in other parts of the time series, known as a motif. Conversely, an unusually high Matrix Profile value (discord) indicates that the given subsequence is unique in the time series and could potentially represent an anomaly.

1.1.6 Anomaly Detection Using Matrix Profile

Given a Matrix Profile P for a time series T, anomaly detection becomes straightforward because the anomalous subsequences have a high distance value compared to the other subsequences [16]. By using an appropriate threshold, we can effortlessly identify the anomalous subsequences in the time series, as illustrated in the figure 2.

In an online scenario, the subsequences on the right-hand side of t_i are unknown (future subsequences). The left matrix profile takes this into account and calculates the distances for a subsequence at time step i based solely on the left-hand-side subsequences. The first few values of the left matrix profile are high

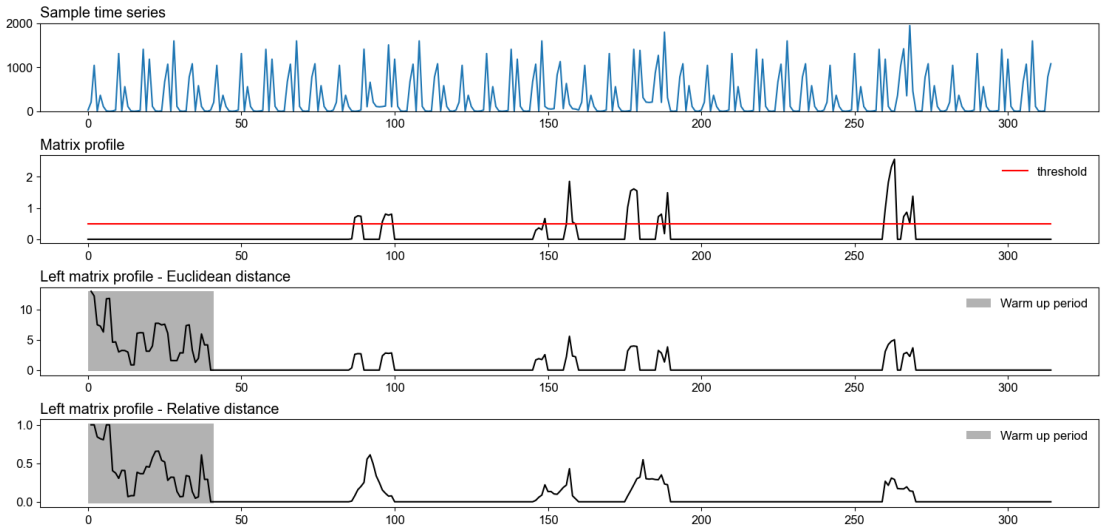


Figure 2: An example of a matrix profile and left matrix profiles using Euclidean distance and relative distance

because there are not many subsequences to compare against during the initial stages. This is known as the warm-up period of the left matrix profile [17].

We can consider the left matrix profile value at time step i as the anomaly score for the data point i. However, the Euclidean distance (Equation 1) has no upper bound, making it challenging to obtain an appropriate anomaly threshold. As mentioned in [17], we can replace the distance measure of the matrix profile algorithm with any suitable distance measure. An ideal scenario would be to use a distance measure that outputs a value bounded between 0 and 1. As shown in RAMP [18], the relative distance measure would be an appropriate choice in this scenario, as it outputs a value between 0 and 1. The relative distance D^R is defined as:

$$D^R(T_{i,m}, T_{j,m}) = min\left(1, \frac{\|T_{i,m} - T_{j,m}\|_1}{\|T_{j,m}\|_1}\right). \tag{2}$$

Figure 2 shows the original time series, matrix profile, and left matrix profiles for both the Euclidean and relative distance measures for the same time series.

1.1.7 Real-time Anomaly Detection Models

Over the past decade, many researchers [11], [19]–[25] have proposed various real-time machine-learning-based anomaly detection models. Although most supervised models have reported superior performance [22]–[25], they are not preferred because most real-world systems do not match their strict criteria. For example,

in many scenarios, it is infeasible for a human operator to label each data point as normal or abnormal, especially when dealing with several hundred to several thousand data streams. Additionally, it is impossible to identify all anomalous situations a priori to train a supervised model. Unsupervised or semi-supervised models are preferred because they can be adapted quickly to any real-world scenario.

There are two main categories of unsupervised/semi-supervised anomaly detection models. The first category involves anomaly detection using possible next-value prediction. An anomaly is flagged if the predicted value deviates significantly from the actual value. Models like HTM [20], Autoencoders [21], Long Short Term Memory (LSTM), and other neural network (NN)-based models like Deep Log [11] follow this approach with high prediction accuracy. However, the main disadvantage of an NN-based approach is that it requires a large amount of training data.

The other category detects abnormal values that do not match pre-configured benign (normal-state) values. The matrix profile model described above falls into this category. Some models use a distance measure like the distance to k nearest neighbors of a value [19], or kernel-based methods like EXPoSE [26] and SVM [27].

1.1.8 Natural Language Understanding

All the logging statements written by software developers are in natural language (primarily English). Therefore, we can leverage Natural Language Understanding (NLU) techniques to extract meaningful information from the log data rather than treating them as mere text. NLU is a subfield of Natural Language Processing (NLP) concerned with enabling machines to interact with human language and allowing machines to extract the meaning of sentences. In recent years, there has been a massive surge in this area, giving rise to transformer-based [28] large language models (LLMs) like BERT [29] and GPT [30]. Some of these models are highly intelligent and even surpass human experts in certain areas [31], [32]. It has been shown that the size of the LLM (number of parameters) is directly proportional to its performance [33]; however, there are some relatively small models fine-tuned for specific tasks that outperform large models [34]. The success of LLMs can be mainly attributed to few-shot learning (conditioning the model with few examples) and zero-shot learning, where we directly prompt instructions [35].

The ability to understand natural language can be leveraged to gain insights

from log data, which has information about system events and behaviors. By applying NLU techniques, we can extract contextual information from log messages rather than treating them as sequences of log keys. This leads to more effective anomaly detection, root cause analysis, and improved interpretability of the models' decisions.

1.2 Motivation

The software industry has widely adopted deploying applications and services on large-scale cloud platforms like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure in recent years [36]. These cloud service systems are required to offer a range of services to millions of users worldwide every day, which makes high service availability crucial. Even minor issues can have significant consequences, and many service providers have put in considerable effort to maintain it [37].

Cloud VM is one such service provided by the cloud vendors, and they use several fault tolerance methodologies to keep the VMs up and running most of the time with minimal service downtime. For example, AWS claims to have "five nines" [38], which means a service availability of 99.999%, allowing at most 26 seconds of downtime per month per VM.

Despite significant effort devoted to quality assurance, cloud service systems continue to face many problems and frequently fail to meet user requests. These problems are often due to computing node (physical server) failures and VM failures within cloud service systems[39], [40]. Such systems typically comprise a vast number of computing nodes that provide processing, network, and storage resources for VM instances.

Accurate failure prediction will reduce VM downtime significantly, but as identified by Lin, Hsieh, Dang, *et al.* [39], the failure prediction of cloud service systems is extremely challenging due to the following reasons,

Complex causes of node failure: Because of the complexity of the cloud architecture, the node (PM) failure may be caused by different software or hardware issues.

Complicated failure indication: Detection of failure of the node is hard and could be indicated by many signs, and one node failure may affect other nodes to fail due to explicit/implicit dependencies among them.

Highly imbalanced failure data: Node fault/failure data are extremely unbalanced, meaning most of the data collected by the nodes will be allocated to

the healthy class, and failure data are very rare because of the low failure rate of the nodes.

Researchers have attempted to achieve high VM availability by building a node failure prediction algorithm to proactively move the failure-prone VMs to a healthy node using live migration. They have used several strategies to achieve this while tackling the issues mentioned earlier. Most of them have used physical server (node) resource usage history data to train an ML model to predict the possible future node failure. Even though these papers claim to have achieved successful node failure prediction, they have left out the most crucial part of any digital system that keeps track of the system state and the events: the logs.

VM live migration takes some time, say x minutes, to move a VM from one node to another, so we need a way to reliably predict the VM failure before the time it takes to move the VM to a healthy node. That is, the failure should be predicted x minutes to VM failure. Most of the papers in this area have ignored this basic fact, and even though the failure prediction may be accurate, the VM may fail during migration due to late failure prediction.

Overall, utilizing log analysis and machine learning to detect failure indicators in logs and predicting VM failures *before the time* it takes to migrate the VM and proactively migrating the VM using the live migration technique is an effective strategy for reducing service downtime and ensuring a seamless user experience.

1.3 Research Questions

1. How to effectively utilize VM and server logs for machine learning (ML) based VM failure prediction?

2. How to develop a generalized VM failure prediction approach using log analysis, enabling its applicability to a wide range of generic VMs?

3. How can the timing of VM failure prediction be optimized to ensure a successful migration, considering the total time required for the VM migration?

1.4 Aims and Objectives

The main aim of this study is to advance the field of virtual machine failure prediction using log analysis for VM fault tolerance, contributing to improved reliability and performance of VMs in server environments such as Cloud Data Centers (CDCs).

These are the main objectives of this study:

- To develop a generalized ML-based prediction approach that leverages key events and indicators present in VM and server logs to predict failures in a variety of VMs.
- To asses the prediction times of the system, compared to the total time required for the VM migration to ensure successful VM migration.
- To evaluate the proposed prediction approach and compare its performance against existing techniques using VM and server log data.

1.5 Scope

1.5.1 In Scope

- **VM failure simulation:** The VM failure log dataset will be collected by simulating VM and PM failures using fault injection techniques because the collection of real-world failure data is infeasible during the project timeframe.
- **VM failure prediction:** The study will focus on developing and evaluating techniques for predicting failures in both VMs and PMs running Ubuntu Server as the host OS, using the logs from the server and the hypervisor (QEMU-KVM).
- **Log analysis:** The research will explore the utilization of logs generated by the server and QEMU-KVM hypervisor to extract log events and system states for failure prediction using an ML approach.
- **Online prediction:** The project will emphasize the development of an online prediction model that can continuously monitor logs to detect failure indicators to predict failures.

1.5.2 Out of Scope

- **Simulation of long-term and cascading VM failures:** This project will emphasize the simulation of only independent failures (like hard disk failure) within a short time window. We did not simulate long-term failure scenarios and cascading failure scenarios, where one failure may lead to another due to time constraints, and their complexity. It is also worth noting

that any method/algorithm that can identify short-term failures and independent failures should also be capable of identifying long-term failures and cascading failures even ahead of time.

- **Integrity of VMs before failure:** We assume that the anomalies (faults) do not cause the VM to corrupt before the failure and the integrity (correctness) of the VMs' disk, memory, and state remain the same as if it were in the normal operation condition. The VM may get corrupted after the failure.

- **Network-related failures:** The research will not specifically address failures related to network components or network infrastructure. It is important to note that in cases where network failure occurs, migration of the VM may not be feasible or effective, as the network failure may impact the migration process itself.

1.6 Outline of the Dissertation

The rest of this dissertation is organized as follows: Section 2 reviews the related work for this research. Section 3 discusses the data collection process. Section 4 presents the design and implementation details of VMFT-LAD. Section 5 evaluates the performance of VMFT-LAD using various metrics and compares it with state-of-the-art anomaly detection models. Section 6 shows how the hyperparameters of our model affect its performance. Section 7 discusses the utility of log anomaly detection on proactive VM fault tolerance and the implications and potential upgrades for our model. Section 8 concludes the paper and presents possible future research directions.

2 Literature Review

2.1 Physical Machine Failure Prediction

In large-scale cloud computing environments with thousands of physical servers, it is inevitable to encounter frequent server failures. According to the studies by Vishwanath and Nagappan [41], and Birke, Giurgiu, Chen, *et al.* [42], approximately 6-8% of all servers experienced at least one hardware issue during a year. Therefore, accurately predicting PM failures is crucial for ensuring cloud system FT.

Numerous studies have been conducted on this topic [39], [43]–[46], and most researchers have utilized PMs' resource usage history data in combination with ML approaches to predict failure. The system administrators label the collected data to train a suitable supervised ML model to predict the failures. For example, the framework proposed by Guan, Zhang, and Fu [43] uses an unsupervised failure detection model using an ensemble of Bayesian models, and the found anomalous data get verified and labeled by system administrators. This labeled data is then used to train a random forest model to classify the server resource usage data as failure-prone or healthy. The framework proposed by Sun, Chakrabarty, Huang, *et al.* [44] uses a CNN model to predict the PM failures. A similar framework proposed by Gao, Wang, and Shen [45] uses a Bi-directional LSTM model.

One notable framework that stands out from the rest is the MING framework, developed by Lin, Hsieh, Dang, *et al.* [39] at Microsoft Research. Its successful deployment in a production cloud environment sets MING apart, demonstrating its practicality and effectiveness. MING uses temporal data, such as performance data, log rate, and OS events, and spatial data, such as the server rack location and load-balancer data, to train a LSTM and Random forest models, respectively, to predict the server failure. MING also has a server ranking system, which will rank all the servers from their failure-proneness. Top k servers can then be selected as the faulty servers.

2.2 Virtual Machine Failure Prediction

When it comes to VM failures, it can be due to either the physical server it is running on or any of the software components involved, such as the host OS, hypervisor, or guest OS. So, physical server failure prediction is a subset of VM failure prediction.

An analysis conducted by Birke, Giurgiu, Chen, *et al.* [42] revealed that 60% of the collected VM failure cases were attributed to physical server failures, while the remaining 40% were caused by other factors. One approach to addressing VM failures is the utilization of redundant VMs [47], [48]. For instance, Scales, Nelson, and Venkitachalam [47] discusses the VMWare VSphere 4.0 VM FT architecture, which employs a redundant VM pattern to replicate the entire execution state of the primary VM through a backup VM on another physical server. Nevertheless, this redundancy strategy can lead to increased costs for cloud service providers, which is not ideal.

An alternative approach involves predicting VM failures and proactively migrating the failure-prone VMs to other physical servers. Similar to the prediction of physical server failures, several studies have focused on VM failure prediction using ML models based on the VM's resource usage history. For instance, Saxena and Singh [49] conducted a study utilizing an ensemble of predictors using Feed-forward Neural Network (FNN), Support Vector Machine (SVM), and Linear Regression (LR) ML models to identify failure-prone VMs and proactively migrate them to a different host.

Although there is a significant body of research on PM/VM failure prediction utilizing resource usage data, the literature on VM failure prediction using log analysis remains relatively scarce. We will discuss the few papers we found that specifically address this topic in the Supplementary Details subsection.

2.3 Virtual Machine Live Migration Time Estimation

VM live migration time estimation/prediction is another aspect of successful VM failure prediction because the failure should be predicted before the time it takes to migrate the VM. If not, there will be migration failures or downtime. In this study, we focus on QEMU-KVM live migration, where the VMs' disk images are located on a network file system, and hence, disk image migration is not required.

There are several studies on KVM live migration time estimation/prediction, where most use a statistical method to calculate the estimated migration time. In this work, a statistical approach is preferable over an ML-based prediction technique because of low resource utilization.

Elsaid, Abbas, and Meinel [50] have conducted a survey on VM live migration cost-modeling, and under section 7 of their paper, they have compared several models for VM migration time estimation. The simplest model they found was $t = as + b$, where s is the VM memory size, and a and b are constants. This shows

that the VM migration time directly depends on the VM memory. However, the study by Nathan, Bellur, and Kulkarni [51] shows several additional parameters affecting pre-copy migration time, namely, VM memory size, page transfer rate, number of unique pages dirtied during each iteration, and the number of skipped pages. The memory size and the transfer rate can be predetermined. The other two parameters depend on the application running on the VM. Thus, to accurately predict the migration time of the VM, we may need to use VM resource usage data.

The discussed literature focuses on the pre-copy live migration approach. In most cases, the pre-copy takes longer to migrate a VM when compared to other methods [7], [8]. Thus, the above-mentioned migration time estimation method is sufficient for this research.

2.4 Supplementary Details

The framework proposed by Nam, Hong, Yoo, *et al.* [4] focuses on predicting the future failure of Virtual Network Functions (VNFs) in a Network Functions Virtualization (NFV) environment built on OpenStack cloud management software [52]. VNF is a VM that runs a network function application such as a firewall or an Intrusion Detection System (IDS). They leverage log data generated by the VNF application and the VM to predict failures. To convert the log data into word embeddings, they employ a NLP technique using the Google Word2Vec library [53]. A CNN model is then used for failure prediction. The Word2Vec-CNN model achieved an overall F1 score of 0.67, predicting VM failure before 5 minutes of the actual failure.

The authors collected the training log data by using a fault injection method to simulate VM failures. Log data is collected at intervals (m minutes), and labels indicating whether the VM failed are collected after a gap time (n minutes). The trained CNN model predicts VM failure before the gap time. However, the authors observed some unexpected failures with no corresponding failure logs. Therefore, VM failure prediction through log analysis may not cover all possible VM failure scenarios.

In their subsequent work [13], the same authors improved their approach to predicting VM and PM failures in a similar NFV environment. Instead of Word2Vec, they employed the Google Bidirectional Encoder Representations from Transformers (BERT) [54] for word embedding and a CNN model for prediction. The BERT-CNN model achieved an F1 score of 0.74, predicting server failure 30 minutes before the actual failure. However, the impact of the prediction models on VMs'

performance is not mentioned in either paper.

Jeong, Van Tu, Yoo, *et al.* [14] proposed a framework to predict paging failure of Virtual Evolved Packet Core (vEPC) in 4G networking. They used VM and server logs and resource usage information to predict VM failure. Unlike the NLP approach in the previous papers, they used the log count (number of occurrences) of specific log types and resource usage data as inputs to an LSTM model for VNF failure prediction. They evaluated the total system throughput with and without the proposed proactive migration system and demonstrated that the framework successfully prevents long-term vEPC failures leading to depleted throughput.

It is important to note that there is a scarcity of research in the area of virtual machine failure prediction using log analysis, as indicated by the limited number of papers identified during the search. This highlights the need for further research and exploration in this specific field to bridge the existing knowledge gap.

3 Data Collection

This section discusses how we collected the log data for testing and evaluating our model, VMFT-LAD. We collected log data from 4 physical machines over a period of 5 months, simulating different failure scenarios. Two physical machines were deployed to simulate VM failures (source servers), and the other two were set up to monitor the activity and collect the log data from the source servers.

3.1 Experimental Testbed

Figure 3 presents the high-level architecture of the testbed setup. The servers had Ubuntu server host OS and QEMU-KVM hypervisor installed. We configured the rSyslog client and rSyslog server in the source and monitoring servers, respectively, to collect and stream log data from the source server. Some of the log files we collected include kernel log, sudo log, systemd log, networkd log, other application logs, and QEMU logs for each VM.

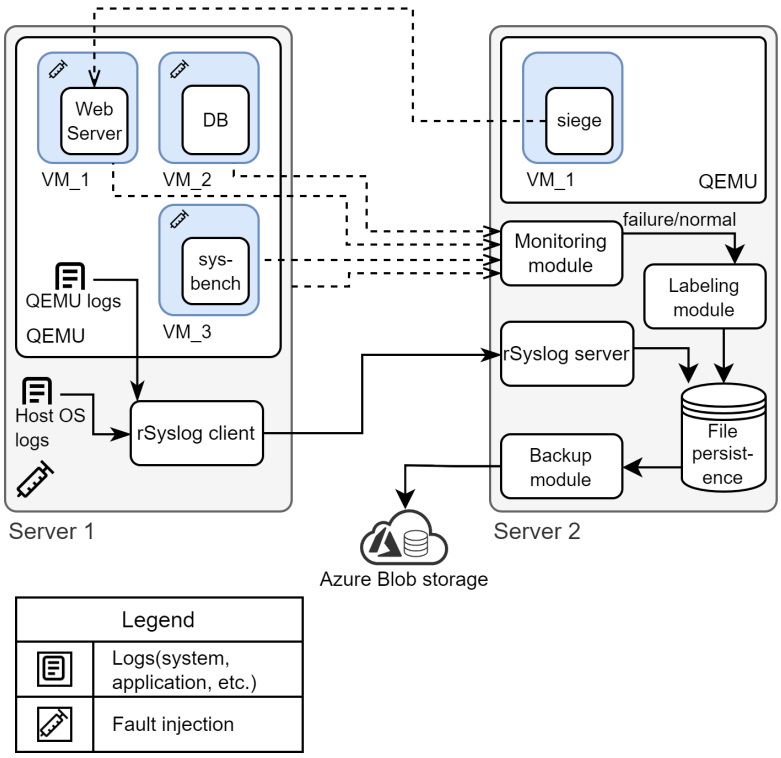


Figure 3: Data collection testbed architecture

In the source server, we configured three VMs, each running real-world and synthetic workloads to simulate typical VM usage scenarios. On the monitoring server, we set up a single VM running the Seige application to perform load

testing on the web server running on the source server, simulating client web requests. Additionally, we implemented three modules: one for monitoring the live/failure status of the VMs and source server, another for collecting timestamps of VM/host failures, and a backup module for pushing the collected log data and labels to Azure blob storage for later access. The monitoring module works using a heartbeat protocol, where the host and VMs send a signal to the monitoring module every 15 seconds via an HTTP request using a CRON (the periodic job scheduler of Linux) task to indicate that they are alive.

3.2 Failure Scenarios

According to [41], 78% of server failures are attributed to hard-disk-related issues, while 5% are caused by memory-module-related problems. The CPU and motherboard are considered the most reliable hardware components in servers [41], and they observed no failures in these components during their study period. When it comes to VM failures, most instances are due to resource over-utilization [25], [42], such as physical machines running out of memory and very high CPU utilization. Based on these facts and the low probability of individual servers failing within a year, we opted to simulate server/VM failures using fault injection techniques rather than waiting for actual failures to occur.

We simulated the following errors in this study to collect log data for the evaluation of VMFT-LAD,

1. **Out of memory (OOM) failure** - The OOM failure was simulated by over-allocating the total memory for VMs by 25% of the host's total physical memory capacity. During normal operation, VMs utilize the host's swap area to manage the over-allocation. We injected faults by stressing the VMs' memory using the stress-ng tool. We consider the VM failure to be the instance where the host OS invokes the OOM-killer to kill the VM process.

2. **Hard disk failure** - We simulated hard disk failure according to [55] by creating a faulty pseudo disk using the Linux SCSI_debug module. Failures were defined as the point where we continuously received unrecoverable read errors for block reads from the faulty disk.

3. **Buffer-IO error** - Buffer IO error happens when there is a problem transferring data between the storage device and memory. Multiple such errors indicate a failure in the disk controller, loose connection, or filesystem cor-

ruption [55]. Similar to hard disk failure, for buffer IO errors, we defined failure as the point where we continuously encountered several errors.

4. **CPU over-allocation** - We simulated VM failure due to CPU over-allocation by over-allocating the total VM vCPUs by 30% of the available host CPU cores. During normal operation, the VMs functioned without any noticeable issues. We used the stress-ng tool to simulate a failure by stressing the CPU on VMs and the host. Failure is defined as the point at which the monitoring module does not receive a heartbeat pulse from the host or the VMs within 18 seconds.

We focused on simulating short-term failures in this study because of the time constraints. If anomaly detection works for short-term failures, it should also work for long-term failure scenarios. Also, the simulated failures are independent failure scenarios, where one failure does not lead to another failure due to the complexity and technical difficulty of simulating such cascading failure scenarios. Studying the real-world failure scenarios of a cloud data center, and evaluating our model on that dataset can be a possible future research direction.

3.3 Data Preprocessing

This section explains the preprocessing steps to process the raw log data into a uniform format before anomaly detection.

The collected raw log data are in different formats, with logging patterns unique to each log file. However, as mentioned in Section 1.1.2 above, they have a common pattern of timestamp and log messages. To convert them to a uniform format, we extract the timestamp and the log message from the different types of log files. After extracting the log message, we remove unnecessary logs, such as CRON logs, that were produced by the VM and host monitoring system, that do not contribute much to the VM failure prediction task. Next, we use the DRAIN-3 [15] log parser, as explained in Section 1.1.2 above, to extract the log template, log template ID (log key) and parameters (values).

The final output of the preprocessing stage is a CSV file with 'timestamp' and 'value' columns containing the timestamp and the log key, for each instance of failure simulation. The failure labels are added to a separate JSON file with the fault injection timestamp and the failure timestamp, similar to Numenta Anomaly Benchmark (NAB) [56].

4 Design and Implementation

This section provides the design and implementation details of VMFT-LAD, an online log anomaly detection model for proactive VM fault tolerance. Fig. 4 illustrates the architecture of our model. It has three main sub-components: an anomaly detection module, a subsequence store, and an adaptive training module to handle anomalous situations.

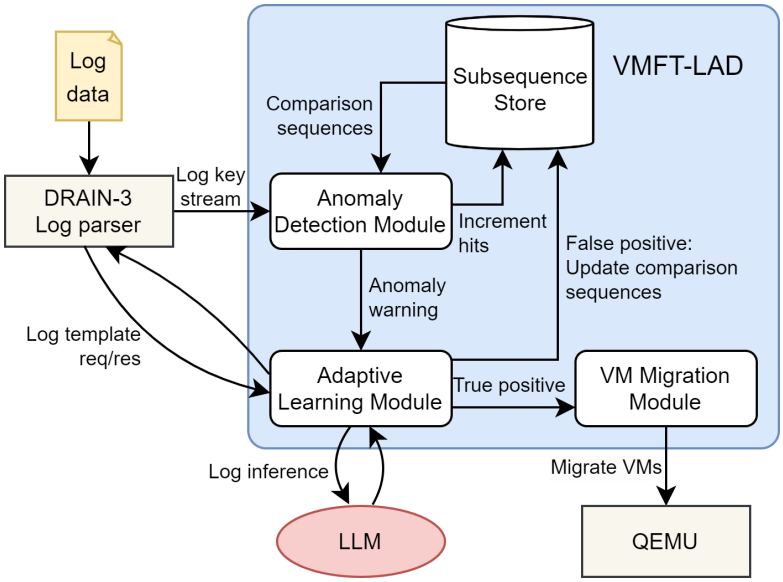


Figure 4: Architecture of VMFT-LAD

Our model runs time-stepped, where for each time step, the anomaly detection module takes in a sequence of log keys of length m, created using the new and past $m-1$ log keys. The anomaly detection module is implemented by utilizing a modified matrix profile model [16]. For each input subsequence, it calculates an anomaly score based on the subsequences stored in the subsequence store. When the anomaly score exceeds the preset threshold θ, the anomaly detection module invokes the adaptive learning module and passes in the anomalous subsequence to handle the anomaly situation. An anomalous situation can occur for two reasons: the anomaly detection module encountered actual fault-related log data, or it can be due to normal but previously unseen logging patterns. While most previous works ignore this false-positive situation, some models [11], [18] use human feedback to adjust their weights to avoid similar false positives in the future. Getting human feedback for data streams of thousands of servers in a cloud data center is slow and impractical, so in this work, we employ a Large Language Model (LLM) to replace human feedback.

The adaptive learning module gets the log template for each log key in the anomalous subsequence and calls an LLM to determine whether the given set of log templates contains failure-related log messages. With that knowledge, it can decide whether to send the migration signal to QEMU or update the comparison sequence store with the current subsequence.

The subsequence store is implemented using max-heap and hash set data structures. Each subsequence is stored alongside the number of hits to that subsequence. The number of hits is the number of times the subsequence gets used during the run, and the max-heap is maintained according to this number. This ensures we access the subsequences in the order of their usage, so more frequently used subsequences get matched first. The hash set is used to prevent the storage of duplicate subsequences.

4.1 Anomaly Detection

The anomaly detection works in a semi-supervised manner, where the matrix-profile-based model learns the normal state log key patterns within a given period. After the initial learning period, the model calculates an anomaly score for each incoming log key subsequence. This method aligns with other anomaly detection models, specifically, those implemented/tested in the Numenta Anomaly Benchmark (NAB) [56].

1. We modify the Matrix Profile similar to RAMP [18] using the relative distance measure as discussed in Section 1.1.6 above instead of the z-normalized Euclidean distance.

2. Unlike RAMP, we only use unique subsequences in the initial training period for comparison, dramatically reducing the model's memory footprint. We also employ a max-heap to order the comparison subsequences according to their frequency of usage so that the most frequently used subsequences will get matched first. This significantly improves the model's speed because the log patterns inherently have frequently repeating patterns.

3. Unlike the Matrix Profile, we are not interested in finding motifs (identification of similar patterns), so we employ a two-level thresholding strategy to increase the efficiency of our model.

4. When the model identifies a subsequence that causes false positives, it suppresses it and updates its comparison set to not give a false positive to a similar subsequence in the future.

Symbol	Description
$T_{i,m}$	Current subsequence
i	Current time step
m	Subsequence length (Window size)
M	Initial learning period (Probationary period)
θ	Anomaly threshold
η	Similarity threshold
C	Subsequence Store
β_i	Anomaly score for time step

Table 2: Symbol legend for algorithms

Table 2 summarizes the symbols used in all the algorithms below.

Algorithm 1 presents our anomaly detection algorithm. For every time step i, the anomaly detection module takes in the current subsequence $T_{i,m}$. If the current time step is in the initial training region, we push the subsequence to the subsequence store (we will discuss the working of the subsequence store later) and return 0 as the anomaly score, as we are currently in the training phase (Lines 1-5). This way, the model learns the unique log patterns in the initial benign region.

Once the training period is over, the model will then calculate the anomaly score for the current time step β_i. First, we assign 1 as a temporary anomaly score (Line 6). Then, we iterate over the benign subsequences in the subsequence store in the order of their frequency of usage. For each subsequence $T_{j,m}$, we calculate the similarity of $T_{j,m}$ and $T_{i,m}$ using the relative distance measure discussed in the section 1.1.6 above (Line 8). If the distance between $T_{j,m}$ and $T_{i,m}$ is less than the similarity threshold η (that means we found a matching benign subsequence), then we increase the number of hits of $T_{j,m}$ and set the current distance as the anomaly score and break out of the loop (Lines 9-12).

Here, we use two different thresholds, the similarity threshold (η) and the anomaly threshold (θ), to define at which distance we define two subsequences as similar (η), and if we do not find a similar subsequence matching the current subsequence in the comparison set, at which minimum distance we define it as an anomaly (θ.) The similarity threshold should be strictly less than the anomaly threshold ($\eta < \theta$) because if not, the model will return all anomalies as similar to some

Also, notice that, unlike the original matrix profile, we did not iterate over all the subsequences to find an exact match; instead, we opted for a close enough solution because we are not interested in finding motifs. Next, if the distance

Algorithm 1 Anomaly detection

```
1: procedure AnomalyDetection(M, i, T_{i,m}, θ, η, C)
2:     if i < M then
3:         C.push(T_{i,m})
4:         return 0
5:     end if

6:     β_i ← 1
7:     for T_{j,m} in C do
8:         Calculate relative_distance for T_{i,m} and T_{j,m}
           with equation 2
9:         if relative_distance < η then
10:            C.increment_hits(T_{i,m})
11:            β_i ← relative_distance
12:            break
13:        else if relative_distance < β_i then
14:            β_i ← relative_distance
15:        end if
16:    end for
17:    if β_i ≥ θ then                                  ▷ Anomaly detected
18:        β_i ← AdaptiveLearning(T_{i,m}, C)
19:    end if
20:    return β_i
21: end procedure
```

for the current subsequence $T_{J,m}$ is less than the previous minimum distance, we update the minimum distance (Lines 13-15).

Once the iteration terminates, we will have the minimum distance for $T_{i,m}$ compared with all $T_{j,m}$ in the subsequence store as the anomaly score for the current time step i. If the anomaly score β_i is less than the anomaly threshold θ, we return it. Otherwise, we must identify whether the anomaly is due to an actual failure-related log or an unanticipated normal log pattern. For that, we call the adaptive learning module (Lines 17 -19).

4.2 Adaptive Learning

The adaptive learning module depicted in the algorithm 2 takes in the anomalous log key subsequence $T_{i,m}$ and it will get the log template corresponding to each log key t_j in $T_{i,m}$ from the log parser, DRAIN-3. Then, we call an LLM to check for failure/fault indicating logs in the log templates.

The LLM acts as a proxy human user and infers the natural language log,

considering the current context to classify whether the given log template contains hardware or software failure/fault-related information.

The following techniques can be used to align the LLM to better understand the context of VM failure-related logs, as understanding the context of VM logs would potentially require domain knowledge a vanilla LLM would not know of unless it is guided in that direction. Few-shot learning: we condition the LLM with few benign log samples to let the LLM understand the context before finally prompting using the current log template to classify it as failure-related or not. Zero-shot learning: we prompt the LLM directly with the log template to classify it without providing any samples. Fine-tuning: we can fine-tune the LLM before prompting to make the model understand the context better with several log samples.

With experimentation, we found that few-shot learning, with only a few benign log template samples, works best for log inference. Also, removing the parameter placeholders like <:*:>, <:NUM:>, <:HEX:> from the log templates greatly improved the classification accuracy of the LLMs we tested. Zero/few-shot prompting is advantageous because it is less costly as there is no training requirement [33], [35], and the LLM will be used as is for inference. Fine-tuning with only benign log data might cause the LLM to lose its inference capability due to overfitting, as this was our experience with fine-tuning LLMs. We will discuss this later in Section 7 below.

Algorithm 2 Adaptive Learning

```
1:  procedure AdaptiveLearning(T_{i,m}, C)
2:      for t_j in T_{i,m} do
3:          log_template ← LogTemplateExtractor(t_j)
4:          faulty_log ← LogInference(log_template)
5:          if faulty_log = True then
6:              return 1                                  ▷ True positive
7:          end if
8:      end for
9:      C.push(T_{i,m})
10:     return 0                                          ▷ False positive
11: end procedure
```

We iterate through the log keys in the anomalous subsequence and then call the LogTemplate extractor function to get the log template corresponding to the log key from the log parser (Line 3). Then, we call an LLM to check whether the current log template contains failure/fault indicators using the LogInference function (Line 4).

The LogInference function works by prompting the LLM with the following prompt with 18 randomly selected benign log templates (for few-shot learning):

```
Classify the given log line into faulty or normal.
Following are some of the normal state logs. Refer to them when
↪  deciding whether the given log template contains a failure
↪  indicator or normal.
Normal state logs:
br0: port entered state,
device promiscuous mode,
...

Command: Classify the given log line as faulty or normal, and give a
↪  short reason in 4-5 words. The faulty log lines should contain a
↪  valid reason for failure. The response should only contain the
↪  result and the reason.
Log line: < Log template here >
Result:
```

The LLM will return a result with a reason for the choice. We also limit the output length by setting the maximum number of new tokens generated to 15. The result is then passed to a function to detect whether it contains "faulty" or not and return True if it does and False if not.

For zero-shot learning, we directly prompt the model with the anomalous log template to classify it into one of three classes: normal, faulty, or unsure. If the model classifies it as faulty, we return True, and if not, False.

If any of the log templates in the anomalous subsequence contains a failure-related log, we take it as a true positive and return 1 as β_i (Lines 5-6). If not, this is a false positive, and to prevent future false positives due to the same subsequence, we update the subsequence store with the current subsequence and return 0 as β_i (Lines 9-10).

This module helps to identify benign log sequences that occur after a long time and may not be present in the initial learning period. These log sequences will appear as anomalous logs but actually are not failure-related. This is a limiting factor in the left matrix profile and also in models like RAMP, where they mark such sequences as anomalous with high confidence.

4.3 Subsequence Store

The subsequence store is responsible for efficient storage and retrieval of log key subsequences with their number of hits. It has a set S that only stores unique

subsequences and is ordered according to the number of hits to the subsequence using the max-heap heapify algorithm. The subsequence store has two primary operations: *push*, which is responsible for inserting a subsequence if it does not exist (Lines 2-5), and *increment_hits*, responsible for incrementing the number of hits of a given subsequence (Lines 8-11). Both operations maintain the heap property by calling the heapify operation after making changes to S.

Algorithm 3 Subsequence Store

```
1:  S ← {}                                              ▷ S is an ordered set
2:  procedure PUSH(T_{i,m})
3:      if T_{i,m} ∉ S then
4:          S ← S ∪ {(id : T_{i,m}, hits : 1)}
5:          Heapify S according to the number of hits of each
            subsequence
6:      end if
7:  end procedure

8:  procedure INCREMENT_HITS(T_{i,m})
9:      t ← S.get(T_{i,m})
10:     t.hits ← t.hits + 1
11:     Heapify S according to the number of hits of each
        subsequence
12: end procedure
```

We implemented the subsequence store using a hash set and a max heap. The max heap is implemented using a list that maintains the max heap property. When inserting a new subsequence, we check the hash set to see if the given subsequence is already present.

The worst-case time complexity of VMFT-LAD for calculating anomaly score for a subsequence $T_{i,m}$ is bounded by $O(lm)$, where l represents the size of the subsequence store (the number of subsequences stored) and m is the subsequence length (window size).

This efficient implementation of the subsequence store, combined with the modified matrix profile algorithm and the adaptive learning module leveraging LLMs, allows VMFT-LAD to effectively detect anomalies in log data and proactively identify potential VM failures with no human intervention. The adaptive learning capability further enhances the model's ability to handle false positives and continuously improve its performance over time.

5 Evaluation

This section presents the evaluation of our model, VMFT-LAD, under the following metrics:

- Receiver Operator Characteristics (ROC) under two criteria
- Area Under the Curve (AUC) for ROC
- Numenta Anomaly Benchmark (NAB) scores under two criteria
- Execution speed

5.1 Evaluation Criteria

We evaluated the performance of the models under two different criteria to check the models' failure detection ability in general and to check for models' early failure indicators detection ability:

5.1.1 Criteria-1: relaxed

Under this criterion, the model is expected to detect failure indicators after the fault injection point and even within the failure region. A True Positive (TP) is counted when the model detects an anomaly in the pre-failure region or after the failure has occurred. This criterion serves as a baseline to verify if the model can correctly identify failures without considering the strict requirement of detecting them before the failure point.

5.1.2 Criteria-2: strict

This criterion is stricter and requires the model to detect failures before the failure point. A True Positive (TP) is counted when the model detects an anomaly in the pre-failure region. This criterion considers the requirement of predicting failure before the failure point, which allows the successful migration of VMs to the destination server without any issues to the VM.

For both criteria, if a model detects an anomaly in the benign region of a dataset (the region before fault injection), it is classified as a False Positive (FP). A False Negative (FN) is defined as the model's failure to identify the anomaly in the pre-failure region or the failure region. Similar to NAB, we do not consider any anomaly flagged by the models during the initial learning period.

5.2 Evaluated Models

We evaluated the VMFT-LAD model with different LLM configurations using both few-shot and zero-shot learning, along with selected four models implemented and evaluated in the Numenta Anomaly Benchmark (NAB) [56] for comparison with the state of the art.

1. VMFT-LAD without feedback: This is the baseline version of our VMFT-LAD model that does not utilize any feedback mechanism from the large language model (LLM).

2. VMFT-LAD with LLM: Several variants of the VMFT-LAD model are evaluated, each incorporating a different LLM and different model conditioning techniques. These LLMs include:

    - GPT-3.5 turbo: This is a state-of-the-art LLM known for its exceptional performance on a wide range of tasks.
    - Falcon 7B, Cyrax 7B, and Emerton Monarch 7B LLMs: High-scoring LLMs based on the Hugging Face Open LLM Leaderboard [57] [1]
    - Bart Large (Zero-Shot): Bart Large is the most popular open-source zero-shot text classifier in the Hugging Face model library [1]

3. HTM: This model uses a different anomaly detection approach based on the Hierarchical Temporal Memory (HTM) architecture [20]. This is a state-of-the-art anomaly detection model and is used in many real-world projects.

4. KNN-CAD: K-Nearest-Neighbours Conformal Anomaly Detection (KNN-CAD) [19] is a K-Nearest Neighbors (KNN) based non-parametric anomaly detection model.

5. EXPOSE: EXPected Similarity Estimation (EXPoSE) [26] is also a non-parametric anomaly detection model based on a kernel function to measure similarity between data points.

6. ARTime: ARTime [58] is based on Adaptive Resonance Theory (ART), and this model outperforms the state-of-the-art model HTM in NAB.

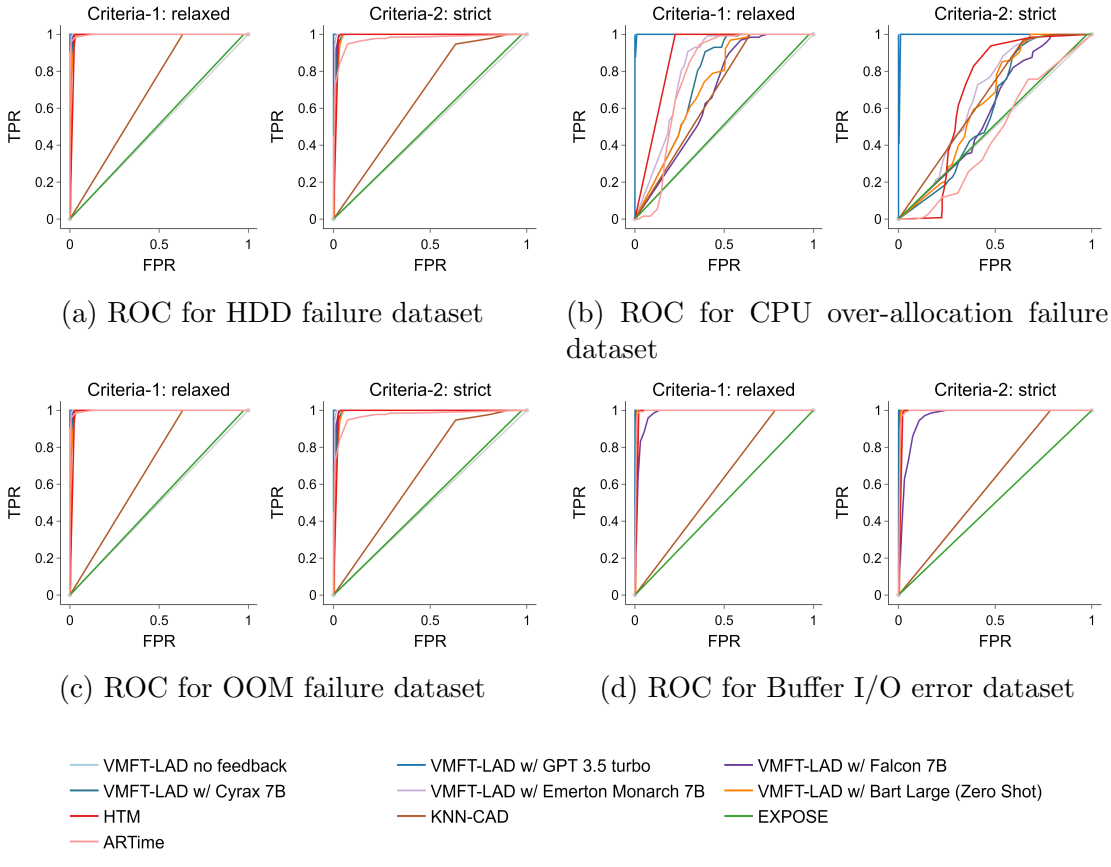
All LLMs used to evaluate VMFT-LAD are conditioned with a small set of randomly selected benign log samples so that the model can better understand

[1] as of February 2024

the VM failure context. The Bart Large (Zero-Shot) model is an exception, as it is specifically evaluated to assess results using an LLM without any conditioning.

5.3 ROC Curve Analysis

Figure 5 presents the Receiver Operator Characteristics (ROC) curves for each dataset using all evaluated models. The ROC curve is a graphical representation of the trade-off between the true positive rate (TPR) and the false positive rate (FPR) at different classification thresholds. A model with better performance will have an ROC curve closer to the top-left corner of the plot, indicating a higher TPR and a lower FPR. Table 3 presents the Area Under the Curve (AUC) values for the ROC curves, which provide a quantitative measure of the overall performance of the models. Higher AUC values indicate better classification performance.



(a) ROC for HDD failure dataset

(b) ROC for CPU over-allocation failure dataset

(c) ROC for OOM failure dataset

(d) ROC for Buffer I/O error dataset

Figure 5: Receiver Operator Characteristics (ROC) curves for each dataset using all evaluated models

	Models	Dataset			
		HDD	OOM	Buffer-IO	CPU
Criteria-1: relaxed	VMFT-LAD no feedback	0.992	0.903	0.993	0.678
	VMFT-LAD w/ GPT 3.5 turbo	0.999	0.999	0.999	0.999
	VMFT-LAD w/ Falcon 7B	0.996	0.965	0.978	0.678
	VMFT-LAD w/ Cyrax 7B	0.992	0.974	0.997	0.754
	VMFT-LAD w/ Emerton Monarch 7B	0.996	0.909	0.993	0.817
	VMFT-LAD w/ Bart Large (Zero Shot)	0.991	0.836	0.993	0.725
	HTM	0.988	0.977	0.99	0.888
	KNN-CAD	0.684	0.626	0.607	0.677
	EXPOSE	0.512	0.499	0.501	0.511
	ARTime	0.997	0.986	0.995	0.780
Criteria-2: strict	VMFT-LAD no feedback	0.987	0.657	0.992	0.588
	VMFT-LAD w/ GPT 3.5 turbo	0.999	0.998	0.999	0.996
	VMFT-LAD w/ Falcon 7B	0.993	0.767	0.962	0.588
	VMFT-LAD w/ Cyrax 7B	0.989	0.899	0.997	0.594
	VMFT-LAD w/ Emerton Monarch 7B	0.995	0.667	0.992	0.669
	VMFT-LAD w/ Bart Large (Zero Shot)	0.987	0.621	0.993	0.634
	HTM	0.987	0.623	0.989	0.680
	KNN-CAD	0.659	0.304	0.607	0.664
	EXPOSE	0.512	0.499	0.501	0.511
	ARTime	0.973	0.546	0.995	0.458

Table 3: Area Under the Curve (AUC) results for ROC curves

5.3.1 HDD Failure Dataset

Under the relaxed criteria (i.e., Criteria-1), the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.999) and ARTime (AUC: 0.997) exhibits near-perfect performance for the HDD failure dataset. The VMFT-LAD with Falcon 7B, Cyrax 7B, and Emerton Monarch 7B LLMs also perform exceptionally well (AUC: 0.992-0.996), closely followed by the VMFT-LAD without feedback (AUC: 0.992) and the HTM model (AUC: 0.988). KNN-CAD (AUC: 0.684) and EXPOSE (AUC: 0.512) show relatively poorer performance for this dataset.

Under stricter Criteria-2, the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.999) continues to exhibit excellent performance. However, the ARTime model (AUC: 0.973) slightly underperforms compared to Criteria-1. All other models performed relatively well for the HDD failure dataset (AUC around 0.987), except for KNN-CAD (AUC: 0.659) and EXPOSE (AUC: 0.512), which showed relatively poorer performance.

5.3.2 CPU Over-allocation Failure Dataset

For the CPU over-allocation failure dataset, the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.999) stands out with an exceptional ROC curve, achieving nearly perfect classification performance under Criteria-1. The HTM model (AUC: 0.888) outperformed all the other models, including VMFT-LAD, with other LLMs. The VMFT-LAD without feedback (AUC: 0.678) and the ARTime model (AUC: 0.780) exhibit moderate performance, while KNN-CAD (AUC: 0.677) and EXPOSE (AUC: 0.511) struggle with this dataset.

Under the stricter Criteria-2, the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.996) maintains its exceptional performance. The HTM model (AUC: 0.680) and VMFT-LAD with Emerton Monarch 7B (AUC: 0.669) LLMs also perform reasonably well. Other models, including VMFT-LAD without feedback (AUC: 0.588) and especially ARTime (AUC: 0.458), struggled with this dataset.

5.3.3 OOM Failure Dataset

In the case of the OOM failure dataset, the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.999) demonstrates outstanding performance under relaxed Criteria-1, closely followed by the ARTime (AUC: 0.986), VMFT-LAD with Cyrax 7B LLM (AUC: 0.974), and the HTM model (AUC: 0.977). The VMFT-LAD with Falcon 7B, with Emerton Monarch 7B, and without feedback also performs reasonably well (AUC around 0.9-0.965). VMFT-LAD with Bart Large (Zero Shot) also exhibits good performance (AUC: 0.836). KNN-CAD (AUC: 0.626) and EXPOSE (AUC: 0.499) struggle with this dataset.

In the case of the OOM failure dataset under stricter Criteria-2, the VMFT-LAD model with GPT 3.5 turbo LLM (AUC: 0.998) continues to demonstrate outstanding performance. The VMFT-LAD with Cyrax 7B LLM (AUC: 0.899) performed well, while the ARTime model (AUC: 0.546) showed very poor performance compared to Criteria-1. KNN-CAD (AUC: 0.304) and EXPOSE (AUC: 0.499) continue to struggle significantly.

5.3.4 Buffer I/O Error Dataset

For the Buffer I/O error dataset under relaxed Criteria-1, all the models achieve near-perfect classification performance, with AUC values above 0.99, except for VMFT-LAD with Falcon 7B LLM feedback (AUC: 0.978), which shows relatively low performance compared to others. KNN-CAD (AUC: 0.607) and EXPOSE (AUC: 0.501) again show very poor performance for this dataset.

Under stricter Criteria-2 for the Buffer-I/O error dataset, most models performed well, with VMFT-LAD models and HTM achieving AUC values above 0.98. The VMFT-LAD with Falcon 7B LLM feedback (AUC: 0.962) showed slightly lower performance, while KNN-CAD (AUC: 0.607) and EXPOSE (AUC: 0.501) again showed very poor performance.

Overall, the VMFT-LAD model with the GPT 3.5 turbo LLM consistently outperforms other models across all datasets and under both evaluation criteria, with near-perfect AUC values, demonstrating its effectiveness in proactive VM fault tolerance using log anomaly detection. The VMFT-LAD without feedback and with other LLMs, such as Cyrax 7B and Emerton Monarch 7B, also exhibit promising performance. The HTM and ARTime models also perform well in certain scenarios, while KNN-CAD and EXPOSE generally struggle across the datasets and evaluation criteria.

5.4 NAB Scores

The Numenta Anomaly Benchmark (NAB) [56] is a benchmark suite designed to evaluate the performance of algorithms for detecting anomalies in streaming data. It provides a standardized framework for comparing the effectiveness of different univariate anomaly detection models. The NAB scores serve as a quantitative measure of a model's ability to identify anomalies while minimizing false positives and false negatives accurately.

NAB calculates three different scores: Standard, Reward Low FP, and Reward Low FN. The Standard score is a balanced score that accounts for both false positives and false negatives. The *Reward Low FP* score emphasizes minimizing false positives, making it suitable for scenarios where false alarms are more costly. Conversely, the *Reward Low FN* score prioritizes minimizing false negatives, which is beneficial when failing to detect an anomaly is more critical.

We modified the original NAB repository by adding our datasets, including the benign dataset and labels according to the NAB specification. The modified NAB repository is available publicly.[2]

Table 4 presents NAB score results under both Criteria-1: relaxed and Criteria-2: strict. Table 5 presents the average false positive rate and average early detection rate (TPR under Criteria-2) for the evaluated models.

Under relaxed Criteria-1, the VMFT-LAD model with GPT 3.5 turbo LLM feedback achieves outstanding NAB scores across all three metrics (around 98),

[2]Modified NAB repository: `https://github.com/CloudnetUCSC/NAB`

	Model	NAB Score		
		Standard	Reward Low FP	Reward Low FN
Criteria-1	VMFT-LAD	98.16	97.77	98.44
	HTM	66.63	61.04	71.64
	KNN-CAD	32.13	-22.05	42.32
	EXPOSE	42.32	37.07	67.33
	ARTime	73.98	56.92	77.89
Criteria-2	VMFT-LAD	90.74	90.36	89.67
	HTM	21.52	16.22	3.13
	KNN-CAD	1.47	-54.94	0.21
	EXPOSE	52.50	18.42	56.17
	ARTime	48.53	26.99	46.55

Table 4: NAB scores for evaluated models

demonstrating its overall effectiveness in accurately identifying anomalies while maintaining a balance between false positives and false negatives. Among the other models, ARTime exhibits the next best performance (standard score: 73), followed by the HTM model (standard score: 71). KNN-CAD and EXPOSE show relatively poorer performance even under this less strict criterion.

Under stricter Criteria-2, which requires anomaly detection before the failure point, the VMFT-LAD model continues to outperform the others with a standard score of 90.74. The other models, however, exhibit a significant drop in performance under this criterion, with very low scores across all three metrics.

Model	False positive rate	Early detection rate
VMFT-LAD	0.02%	96.28%
HTM	0.07%	62.08%
KNN-CAD	0.74%	76.58%
EXPOSE	0.39%	85.13%
ARTime	0.24%	78.25%

Table 5: Average False Positive rate and Early Detection rate at the best threshold

The results in Table 5 present the average false positive rate and average early detection rate, which is the true positive rate under the stricter Criteria-2 for VMFT-LAD model with GPT 3.5 LLM feedback and other evaluated models. The results are calculated using the best threshold for each model identified by the NAB optimizer across all datasets, including the benign dataset. VMFT-LAD shows the lowest false positive rate (0.02%) and the highest early failure indicator detection rate (96.28%).

These results highlight our model's effectiveness in proactive VM fault tolerance using log anomaly detection, which is the early detection of anomalies before failures occur while minimizing the false positives that may lead to service degradation due to unnecessary VM migrations.

5.5 Model Execution Time Analysis

Fig. 6 presents the average execution time required by each model to process a single record and determine whether it is anomalous or not. This metric is essential in determining the models' practicality for real-time anomaly detection.

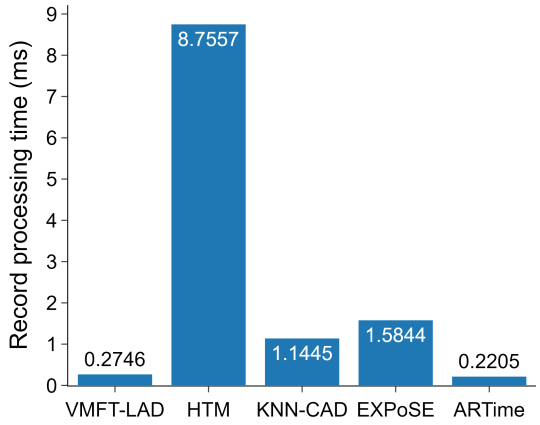


Figure 6: Average execution time to process a record (Lower the better)

The VMFT-LAD without LLM feedback and ARTime models exhibit impressive average execution times. KNN-CAD and EXPOSE have moderate execution times, and the HTM model has a comparatively higher average execution time.

Following are the actual log rates observed in the collected server log data: The average time difference between two consecutive records is 163.966 milliseconds, and The average time between all records is 6.053 seconds. Even the slowest model, HTM, with an average execution time of 8.7557 milliseconds, can comfortably process records at these log generation rates.

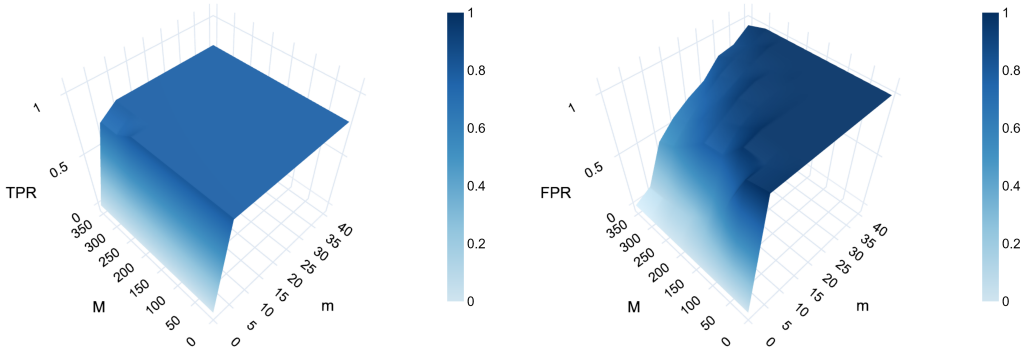
All the evaluated models demonstrate sufficient computational efficiency to handle online anomaly detection in our server environment. However, for scenarios with exceptionally high log generation rates, the VMFT-LAD model without LLM feedback and ARTime can be the most suitable choice due to their sub-millisecond execution times.

6 Hyperparameter Tuning

In this section, we show how the hyperparameters of our model VMFT-LAD affect its performance.

6.1 Training Period and Subsequence Length

Fig. 7 illustrates the impact of varying the training period (M) and subsequence length (m) on the True Positive Rate (TPR) and False Positive Rate (FPR) of the anomaly detection process.



(a) Impact of the training period (M) and subsequence length (m) on True Positive Rates

(b) Impact of the training period (M) and subsequence length (m) on False Positive Rates

Figure 7: Impact of hyperparameter change on anomaly detection performance

Fig. 7(a) depicts the effect of M and m on the TPR. As evident from the plotted surface, increasing the training period length (M) has a minimal impact on the TPR; however, as depicted by Fig. 7(b), increasing M reduces the FPR, which is the expected behavior, because the model will have a larger benign context from the training data.

The subsequence length (m) plays a significant role in determining the TPR. For smaller values of m, the TPR is lower, indicating that the model may struggle to capture the anomalous patterns when considering shorter subsequences. As m increases, the TPR steadily improves, reaching its maximum value for $m \geq 4$. This behavior is expected, as longer subsequences provide more contextual information, allowing the model to identify anomalies within the time series data effectively.

6.2 Similarity Threshold

Fig. 8 presents the impact of the similarity threshold (η) on the record processing time of VMFT-LAD. The results were measured by running VMFT-LAD on a large benign dataset with over 6000 data points to get an average execution time to process a single data point. We set $\theta \leftarrow 0.5$, $M \leftarrow 150$, $m \leftarrow 4$ for the evaluation. As defined in the section 4.1 above, $\eta < \theta$, so we set the η to range from 0 to 0.45 with a step size of 0.05. The plot shows that the record processing time decreases (execution speed increases) when η approaches θ.

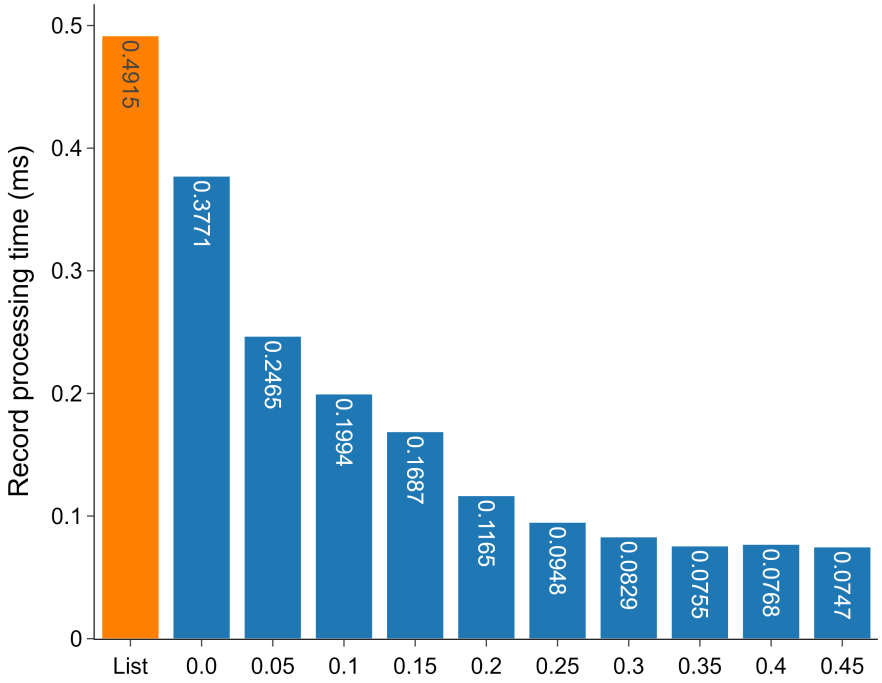


Figure 8: Impact of the similarity threshold (η) over the average record processing time. The average record processing time using a list-based implementation is added for comparison

η does not affect the TPR or the FPR of the model because they are determined by the anomaly threshold θ. We also included the average record handling time for the regular list-based implementation of the subsequence store. The execution speed of VMFT-LAD with the Max-heap-based subsequence store is faster compared to the list-based implementation, even when η is 0 (searches for an exact match with the minimum possible distance). This is because the max-heap implementation searches from the order of the frequency of subsequences, while the list-based implementation does a linear search.

From the results, we can clearly see that choosing the similarity threshold closer

to the anomaly threshold is better for the execution speed of the model. The max-heap-based implementation is 84.63% faster than the list-based implementation. But, even with the list-based subsequence store implementation, VMFT-LAD is faster than some NAB detectors.

7 Discussion

In this section, we discuss the utility of log-based anomaly detection in proactive VM fault tolerance via live migration, how our solution has addressed all our research questions, and our experience with fine-tuning LLMs for failure-related log identification.

7.1 Utility of Anomaly Detection in Proactive VM Fault Tolerance

The effectiveness of proactive fault tolerance in virtual machine environments relies on accurately predicting failures well in advance, allowing sufficient time for VM migration before the failure occurs. In this section, we evaluate the failure prediction capabilities of our model VMFT-LAD compared to other evaluated models to analyze their compatibility with the VM migration times observed in our server environment.

Model	Dataset			
	HDD	OOM	Buffer-IO	CPU
VMFT-LAD	15.651	3.033	13.338	14.535
HTM	15.675	1.535	13.354	8.462
KNN-CAD	15.901	2.012	12.073	12.555
EXPOSE	15.658	3.136	11.864	23.959
ARTime	15.674	4.057	13.337	13.120

Table 6: **Average detection time before failure (minutes)**

Table 6 presents the average detection time of faults before the failure for each model across all our datasets. The results indicate that our model VMFT-LAD achieves the earliest average detection times, ranging from 3.033 minutes for the OOM dataset to 15.651 minutes for the HDD dataset. This early detection capability is crucial for enabling timely migration of VMs before failures occur. All the models performed relatively well, except for the OOM dataset, where all the models seemed to struggle to identify the failure early. This may be due to a lack of early pre-failure indicators in the dataset for OOM failures.

To assess the feasibility of proactive migration, we compare the failure prediction times with the actual VM migration times observed. Fig. 9 illustrates the total migration time for VMs of different sizes, ranging from 1 GB to 20 GB, using three different migration techniques: Vanilla post-copy, Vanilla pre-copy,

and XBZRLE compression enabled pre-copy [59]. We ran Memcached [60] in the VMs when collecting migration time data (Memcached is a high-performance in-memory caching solution for databases). We allowed Memcached to use up to 80% of the VM memory and configured the Memaslap load generation tool to generate the necessary database load to simulate real-world scenarios. We chose Memcached because it is a real-world unified workload that is CPU, memory, and I/O intensive.

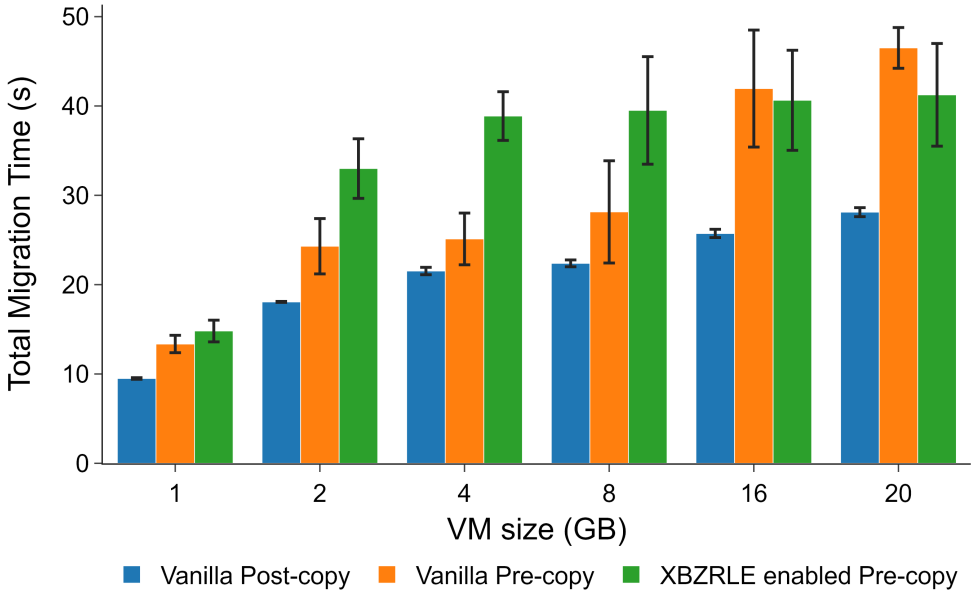


Figure 9: The total migration time for migrating VMs with different v-RAM sizes

We can see that across all VM sizes, the post-copy technique exhibits the lowest migration times. The XBZRLE compression helps to reduce migration times in pre-copy when the VM sizes are relatively large ($\geq$16 GB).

Fig. 10 shows the downtime experienced by VMs during migration. The downtime is the period during which the VM is paused to copy the final states of the VM to the destination, and it is essential to minimize this duration to maintain the quality of service. All the migration methods across all VM sizes show relatively low downtimes ($\tilde{4}$0-255 ms). Specifically, the post-copy technique demonstrates the lowest downtimes ($\tilde{1}$0-45 ms) across all VM sizes.

Comparing the failure prediction times from Table 6 with the migration times shown in Fig. 9, it becomes evident that the models, particularly the VMFT-LAD, provide sufficient lead time to facilitate proactive VM migration before failures occur. For instance, even in the case of the OOM dataset, where the VMFT-LAD achieves an average detection time of 3.033 minutes, the migration time for a 20

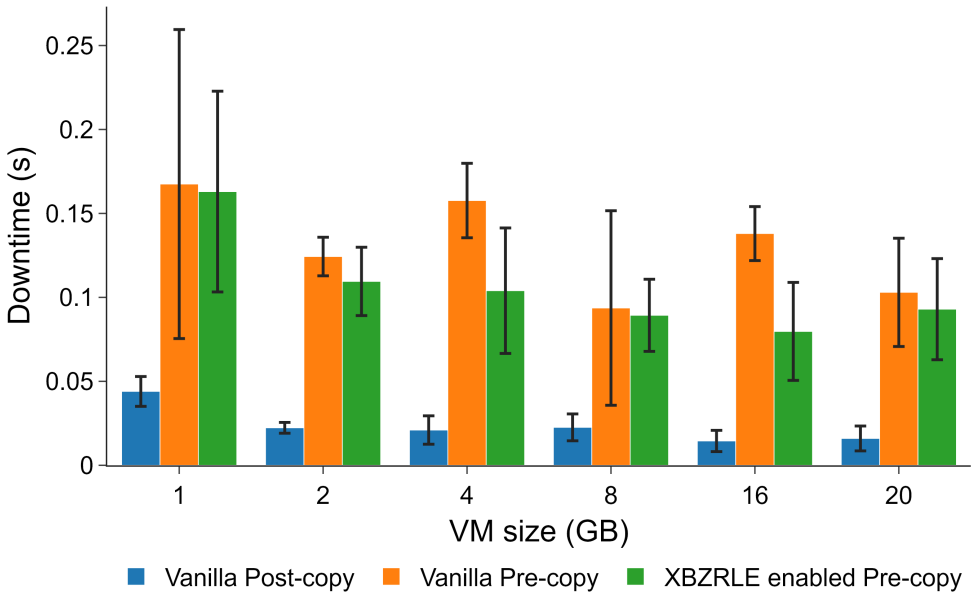


Figure 10: The downtime for migrating VMs with different v-RAM sizes

GB VM using the pre-copy technique is around 45 seconds, migrating one large VM is sufficient to avert the OOM failure of the VMs running in the server.

As evident from all the above results, our solution has addressed all the Research Questions (RQs). RQ1 is solved with our implemented model, VMFT-LAD. RQ2 and RQ3 are satisfied by the above comprehensive evaluation of our model and the VM migration time analysis of our test environment.

All the results highlight the effectiveness of VMFT-LAD in enabling proactive fault tolerance through timely VM failure prediction and migration. The high early detection rate, low FPR, and early detection times, combined with the migration techniques available, ensure that VMs can be migrated to alternative hosts before failures occur, minimizing service disruptions due to failure.

7.2 Exploring LLM Training Paradigms

Our proposed model, VMFT-LAD, operates semi-supervised and quickly learns the normal state logging patterns of the system, requiring as few as 50 data points. It identifies anomalies at a speed comparable to or, in some cases, faster than state-of-the-art approaches. By integrating Large Language Model (LLM) feedback, VMFT-LAD reduces false positives and continuously adapts to changing logging patterns, eliminating the need for human intervention in the VM failure prediction process.

For the LLM feedback, we evaluated several popular and high-performing

LLMs using zero-shot classification and few-shot learning with a few normal state logs. These approaches gave promising classification results. Additionally, we attempted to fine-tune the distilBert LLM using Low-Rank Adaptation (LoRA), utilizing only a subset of the benign logs, adhering to our criterion of only using normal state logs for training. However, the fine-tuned model did not perform well in identifying failures and classified most failure logs as normal.

8 Conclusions

In this work, we presented VMFT-LAD, a novel semi-supervised log anomaly detection model for proactive VM fault tolerance. By combining the efficiency of the Matrix Profile [16], with the log inference capability of large language models (LLMs), VMFT-LAD addresses the limitations of traditional supervised approaches and enables early detection of potential failures, including unforeseen fault types, while continuously adapting to changing log patterns with limited human intervention. Our comprehensive evaluation of VMFT-LAD on several datasets exemplifies its superiority over state-of-the-art real-time anomaly detection models, demonstrating a high early detection rate of 96.28% while maintaining a low false positive rate of 0.02%. VMFT-LAD also demonstrates superior results in the Numenta Anomaly Benchmark (NAB) [56], scoring a standard score of 90.74 under the criterion of predicting failures before the failure point.

However, there is still scope for further improvements and future research directions. One is to test VMFT-LAD in a broader range of failure scenarios and real-world data from diverse cloud environments to further validate its performance and generalization capabilities. Additionally, incorporating multivariate time series data sources, such as resource usage data, alongside log data could potentially enhance the accuracy and robustness of the failure prediction process.

References

[1] R. Jhawar and V. Piuri, "Fault tolerance management in iaas clouds," in *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*, IEEE, 2012, pp. 1–6.

[2] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 252–257. DOI: 10.1109/PDP.2009.31.

[3] A. Polze, P. Tröger, and F. Salfner, "Timely virtual machine migration for pro-active fault tolerance," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, 2011, pp. 234–243. DOI: 10.1109/ISORCW.2011.42.

[4] S. Nam, J. Hong, J.-H. Yoo, and J. W.-K. Hong, "Virtual machine failure prediction using log analysis," in *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, IEEE, 2021, pp. 279–284.

[5] C. Clark, K. Fraser, S. Hand, *et al.*, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005, pp. 273–286.

[6] M. Hines and K. Gopalan, "Post-copy based live virtual machine migration using pre-paging and dynamic self-ballooning," Mar. 2009, pp. 51–60. DOI: 10.1145/1508293.1508301.

[7] A. Shribman and B. Hudzia, "Pre-copy and post-copy vm live migration for memory intensive applications," in *Euro-Par 2012: Parallel Processing Workshops: BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, Rhodes Islands, Greece, August 27-31, 2012. Revised Selected Papers 18*, Springer, 2013, pp. 539–547.

[8] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, vol. 43, no. 3, pp. 14–26, 2009.

[9] R. W. Ahmad, A. Gani, M. Shiraz, F. Xia, S. A. Madani, *et al.*, "Virtual machine migration in cloud data centers: A review, taxonomy, and open research issues," *The Journal of Supercomputing*, vol. 71, no. 7, pp. 2473–2515, 2015.

[10] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE international conference on web services (ICWS)*, IEEE, 2017, pp. 33–40.

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[12] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, IEEE, 2016, pp. 859–864.

[13] S. Nam, J.-H. Yoo, and J. W.-K. Hong, "Vm failure prediction with log analysis using bert-cnn model," in *2022 18th International Conference on Network and Service Management (CNSM)*, IEEE, 2022, pp. 331–337.

[14] S. Jeong, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Proactive live migration for virtual network functions using machine learning," in *2021 17th International Conference on Network and Service Management (CNSM)*, IEEE, 2021, pp. 335–339.

[15] IBM, *Drain3*, `https://github.com/IBM/Drain3`, [Accessed 30-03-2024], 2020.

[16] C.-C. M. Yeh, Y. Zhu, L. Ulanova, *et al.*, "Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *2016 IEEE 16th international conference on data mining (ICDM)*, Ieee, 2016, pp. 1317–1322.

[17] S.-Y. Lan, R.-Q. Chen, and W.-L. Zhao, "Anomaly detection on it operation series via online matrix profile," *arXiv preprint arXiv:2108.12093*, 2021.

[18] J. D. Herath, C. Bai, G. Yan, P. Yang, and S. Lu, "Ramp: Real-time anomaly detection in scientific workflows," in *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, 2019, pp. 1367–1374.

[19] E. Burnaev and V. Ishimtsev, "Conformalized density-and distance-based anomaly detection in time-series data," *arXiv preprint arXiv:1608.04585*, 2016.

[20] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[21] H. Xu, W. Chen, N. Zhao, *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 187–196.

[22] D. Liu, Y. Zhao, H. Xu, *et al.*, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proceedings of the 2015 internet measurement conference*, 2015, pp. 211–224.

[23] D. T. Shipmon, J. M. Gurevitch, P. M. Piselli, and S. T. Edwards, "Time series anomaly detection; detection of anomalous drops with limited features and sparse examples in noisy highly periodic data," *arXiv preprint arXiv:1708.03665*, 2017.

[24] H. Ren, B. Xu, Y. Wang, *et al.*, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 3009–3017.

[25] D. Saxena and A. K. Singh, "Ofp-tm: An online vm failure prediction and tolerance model towards high availability of cloud computing environments," *The Journal of Supercomputing*, vol. 78, no. 6, pp. 8003–8024, 2022.

[26] M. Schneider, W. Ertel, and F. Ramos, "Expected similarity estimation for large-scale batch and streaming anomaly detection," *Machine Learning*, vol. 105, pp. 305–333, 2016.

[27] J. Ma and S. Perkins, "Time-series novelty detection using one-class support vector machines," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, IEEE, vol. 3, 2003, pp. 1741–1745.

[28] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

[30] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[31] X. Luo, A. Rechardt, G. Sun, *et al.*, "Large language models surpass human experts in predicting neuroscience results," 2024. [Online]. Available: `https://api.semanticscholar.org/CorpusID:268253470`.

[32] D. Van Veen, C. Van Uden, L. Blankemeier, *et al.*, "Clinical text summarization: Adapting large language models can outperform human experts," *Research Square*, 2023.

[33] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[34] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[35] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[36] J. Bulao. "How many companies use cloud computing in 2023? all you need to know." (2023), [Online]. Available: `https://techjury.net/blog/how-many-companies-use-cloud-computing/#gref` (visited on 01/22/2023).

[37] J. Khalili. "Here's how much cash google lost due to last week's outage." (2020), [Online]. Available: `https://www.techradar.com/news/google-blackout-saw-millions-in-revenue-vanish-into-thin-air` (visited on 01/28/2023).

[38] R. Reynolds. "Achieving "five nines" in the cloud for justice and public safety." (2020), [Online]. Available: `https://aws.amazon.com/blogs/publicsector/achieving-five-nines-cloud-justice-public-safety/` (visited on 05/12/2023).

[39] Q. Lin, K. Hsieh, Y. Dang, *et al.*, "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*, 2018, pp. 480–490.

[40] A. Lawrence. "2022 outage analysis finds downtime costs and consequences worsening as industry efforts to curb outage frequency fall short." (2022), [Online]. Available: `https://uptimeinstitute.com/about-ui/press-releases/2022-outage-analysis-finds-downtime-costs-and-consequences-worsening` (visited on 01/28/2023).

[41] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 193–204.

[42] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, "Failure analysis of virtual and physical machines: Patterns, causes and characteristics," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 1–12. DOI: 10.1109/DSN.2014.18.

[43] Q. Guan, Z. Zhang, and S. Fu, "Ensemble of bayesian predictors and decision trees for proactive failure management in cloud computing systems.," *J. Commun.*, vol. 7, no. 1, pp. 52–61, 2012.

[44] X. Sun, K. Chakrabarty, R. Huang, *et al.*, "System-level hardware failure prediction using deep learning," ser. DAC '19, Las Vegas, NV, USA: Association for Computing Machinery, 2019, ISBN: 9781450367257. DOI: 10.1145/3316781.3317918. [Online]. Available: https://doi.org/10.1145/3316781.3317918.

[45] J. Gao, H. Wang, and H. Shen, "Task failure prediction in cloud data centers using deep learning," *IEEE transactions on services computing*, vol. 15, no. 3, pp. 1411–1422, 2020.

[46] R. S. A. K. Abid, M. M. Taher, and S. A. R. Ahmed, "Task failure prediction in cloud data centers using deep learning," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 14, no. 2, pp. 716–723, 2023.

[47] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30–39, 2010.

[48] J.-B. Decourcelle, T. D. Ngoc, B. Teabe, and D. Hagimont, "Fast vm replication on heterogeneous hypervisors for robust fault tolerance," in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 15–28.

[49] D. Saxena and A. K. Singh, "Ofp-tm: An online vm failure prediction and tolerance model towards high availability of cloud computing environments," *The Journal of Supercomputing*, vol. 78, no. 6, pp. 8003–8024, 2022.

[50] M. E. Elsaid, H. M. Abbas, and C. Meinel, "Virtual machines pre-copy live migration cost modeling and prediction: A survey," *Distributed and Parallel Databases*, vol. 40, no. 2-3, pp. 441–474, 2022.

[51] S. Nathan, U. Bellur, and P. Kulkarni, "Towards a comprehensive performance model of virtual machine live migration," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 288–301.

[52] OpenStack. "What is openstack?" (2023), [Online]. Available: `https://www.openstack.org/software/` (visited on 05/17/2023).

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013. arXiv: `1301.3781 [cs.CL]`.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: `1810.04805 [cs.CL]`.

[55] N. Georgoulopoulos, A. Hatzopoulos, K. Karamitsios, I. M. Tabakis, K. Kotrotsios, and A. I. Metsai, "Investigation and simulation of hardware errors in kernel logs of linux-based server systems," in *2021 6th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, IEEE, 2021, pp. 1–7.

[56] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[57] N. L. Edward Beeching Sheon Han. "Open llm leaderboard." (2023), [Online]. Available: `https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard` (visited on 04/09/2024).

[58] M. Hampton. "Artimenab." (2021), [Online]. Available: `https://github.com/markNZed/ARTimeNAB.jl` (visited on 10/29/2023).

[59] A. Shribman and B. Hudzia, "Pre-copy and post-copy vm live migration for memory intensive applications," in *Euro-Par 2012: Parallel Processing Workshops: BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, Rhodes Islands, Greece, August 27-31, 2012. Revised Selected Papers 18*, Springer, 2013, pp. 539–547.

[60] B. Fitzpatrick, *Memcached: A distributed memory object caching system*. [Online]. Available: `https://memcached.org/`.