IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 November 2024, accepted 21 November 2024, date of publication 26 November 2024, date of current version 9 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3506833

RESEARCH ARTICLE

Virtual Machine Proactive Fault Tolerance Using Log-Based Anomaly Detection

PRATHEEK SENEVIRATHNE, SAMINDU COORAY, JEROME DINAL HERATH, AND DINUNI FERNANDO

School of Computing, University of Colombo, Colombo 00700, Sri Lanka

Corresponding author: Dinuni Fernando (dkf@ucsc.cmb.ac.lk)

ABSTRACT Virtual Machine (VM) fault tolerance ensures high availability in cloud computing environments. Proactive fault tolerance strategies avert service disruptions by detecting potential failures before they occur and migrating the VMs to healthy hosts. In this paper, we propose Virtual Machine Proactive Fault Tolerance using Log-based Anomaly Detection (VMFT-LAD), a semi-supervised, real-time log anomaly detection model capable of detecting failures ahead of time to provide effective 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 identify potential VM failures early, while minimizing false positives. Our improved Matrix Profile enables VMFT-LAD to continuously learn and identify potential failures, including unforeseen fault types, with minimal human intervention. Additionally, its semi-supervised nature eliminates the need for labeled failure data. Extensive evaluations on several datasets, using two distinct criteria to validate anomaly detection and early failure detection capabilities, demonstrate VMFT-LAD's outstanding performance. VMFT-LAD achieves a Numenta Anomaly Benchmark (NAB) standard score of 90.74 for predicting failures in advance, with a high early detection rate of 96.28% and a low false positive rate of 0.02%, enabling accurate and timely VM migration before failures occur.

INDEX TERMS Adaptive learning, anomaly detection, cloud computing, fault tolerance, large language models, log analysis, matrix profile, natural language processing, proactive migration, virtual machines.

I. INTRODUCTION

Cloud computing has revolutionized how computing resources are provisioned and consumed. Virtual Machines (VMs) are a fundamental component of cloud computing environments, enabling on-demand resource allocation, scalability, and efficient utilization of the underlying hardware. Ensuring high availability and fault tolerance for VMs is crucial to maintain service level agreements (SLAs) and provide uninterrupted services to end-users, as most cloud service providers rely on VMs [1].

VM migration plays a major role in facilitating proactive fault tolerance, providing an unnoticeable transfer of running VMs from one server to another on a millisecond to microsecond scale. This technique is not only employed for fault tolerance but also for load balancing, resource consolidation, and administrative tasks. There are two approaches for VM migration for fault tolerance: proactive and reactive. Proactive fault tolerance involves identifying potential failures before they occur and preemptively migrating the VMs to healthy hosts, averting service disruptions. Whereas reactive fault tolerance strategies start VM migration after a failure has been detected, as a response, which may lead to temporary service outages or even complete loss of the VM state [2], [3], [4].

Proactive fault tolerance is particularly beneficial as it minimizes downtime and data loss by preemptively addressing potential issues. It can identify early warning signs of impending failures by monitoring VMs in real-time, allowing for timely VM migration to healthy hosts [1], [2], [5].

VM failures can originate from various sources, including hardware failures, software/firmware issues, network

The associate editor coordinating the review of this manuscript and approving it for publication was Hadi Tabatabaee Malazi.

failures, resource overutilization, and configuration errors. They can cause service disruption and data loss. For instance, server clusters in Google data centers experience around 1000 server failures during their first year due to hard drive failures, overheating issues, network failures and other reasons [6]. Most importantly, [6] states there is a 50% chance that the cluster overheats, taking down most servers in less than 5 minutes. This further signifies the importance of VM proactive fault tolerance in minimizing service disruptions due to failures.

Logs, which contain system events and states, can serve as valuable sources of failure indicators for proactive VM fault tolerance. However, manually analyzing logs to identify potential failures is impractical, especially in large-scale cloud environments. This is where machine learning (ML) techniques come into play, automating the process of failure prediction based on log data analysis.

Although log data may not capture all possible VM failure scenarios, logs can capture specific hardware and software faults with greater accuracy than resource usage metrics such as CPU, I/O, and memory utilization, which can fluctuate and produce false positives. Log data record the actual occurrence of errors rather than inferring problems from resource consumption patterns, making logs a more reliable indicator of system health [7].

The existing work [8], [9], [10], [11] on VM failure prediction primarily utilizes supervised ML models trained on labeled data, which requires significant human effort. Also, they may struggle to adapt to changing environments in real time or may not even identify failure types on which they were not trained. Additionally, these studies have largely focused on physical machine resource usage history and physical component health data such as error rates, overlooking the potential insights provided by VM and server logs. Moreover, previous work [8], [9] has often been limited to specific types of VMs, such as Virtual Network Functions (VNFs), and has incorporated application-specific data, overlooking the more commonly used generic VMs in cloud computing environments. Furthermore, it is imperative to identify the failure early, at least before the time it takes to move out the VM. Still, the previous studies have disregarded evaluating this aspect of their work.

An ideal proactive VM fault tolerance framework should meet the following key requirements (Ahmad et al. [12] and Lin et al. [13]).

- Early identification of failure indicators to facilitate timely migration.
- Adaptability to changing environments, software updates, and hardware changes.
- Ability to identify any unforeseen failure or fault types.
- Capability to work with highly imbalanced data, where failure data are very rare compared to healthy states.
- Minimization of false positives (which can lead to unnecessary migrations and service disruptions, increasing costs) and false negatives (resulting in service disruptions due to failures).
- Ability to work independently, with little to no human intervention.

In this work, we propose Virtual Machine Proactive Fault Tolerance using Log-based Anomaly Detection (VMFT-LAD), a semi-supervised log anomaly detection model that addresses these research gaps and aims to address the above objectives. The key contributions of VMFT-LAD are as follows:

1) VMFT-LAD is a real-time semi-supervised log anomaly detection model that leverages hypervisor and server logs to identify VM failures ahead of time (prior to migration failures), with the capability to continuously adapt to changing log patterns at runtime with minimal human intervention.
2) This work establishes a diverse log dataset consisting of various VM failure scenarios.
3) We evaluated VMFT-LAD through rigorous comparison with state-of-the-art real-time and deep learning-based anomaly detection models utilizing anomaly detection benchmarks, validating its effectiveness in terms of both early failure detection capability and generalizability across any anomaly situation.
4) We validate the model's early failure detection capability, by using detection lead time compared with real-world VM migration data, ensuring sufficient time for successful migration before failure occurs.

Our model analyzes logs utilizing a heap-based in-memory storage mechanism combined with a modified Matrix Profile, a fast and efficient time series data analysis technique. It leverages the power of large language models (LLMs) to understand the content of logs, continuously adapt to changing log patterns, and identify potential failure indicating logs among anomalous logs, including unforeseen fault types with minimal human intervention. This approach enables VMFT-LAD to operate without needing labeled failure data, making it suitable for dynamic cloud environments.

The remainder of this paper is organized as follows: Section II provides the background details on VM live migration, system logs, log parsing, and anomaly detection techniques. Section III discusses the data collection process. Section IV presents the design and implementation details of VMFT-LAD. Section V evaluates the performance of VMFT-LAD using various metrics and compares it with state-of-the-art anomaly detection models. Section VI shows how the hyperparameters of our model affect its performance. Section VII discusses the utility of log anomaly detection in proactive VM fault tolerance and the implications of our model. Section VIII reviews the related work. Section IX concludes the paper.

II. BACKGROUND

This section provides background details related to VMFT-LAD.

A. VIRTUAL MACHINE LIVE MIGRATION

VM Live Migration is an effective technique for ensuring high availability and uninterrupted service delivery in Cloud Data Centers (CDCs). Live migration allows for the seamless transfer of a running VM from one physical host machine (source) to another (destination), with minimal disruption to its operation [14]. This process involves copying the VM's memory state, CPU context, and virtualized devices to the destination server while the VM and the applications running in it continue to execute. By using live migration, cloud administrators can proactively address potential VM failures or resource constraints.

This proactive approach to fault tolerance contrasts with reactive methods that handle failure after detecting it, which may lead to temporary service outages or even complete loss of the VM state. Live migration for proactive fault tolerance works by identifying and anticipating potential hardware or software issues with the host machine by using a predictive algorithm, such as the one we propose to identify imminent hardware/software problems, to trigger the live migration to move the VM to a healthy host, preventing service disruption due to a failure. Such a failure of VMs running in a cloud environment may lead to customer dissatisfaction, SLA violations, or even the loss of a critical workload [2], [5].

While live migration helps avert VM failure, the live migration process itself might fail before the migration completes, leading to a complete loss of the VM state. Live migration fault tolerance techniques [3], [15], which use a checkpointing mechanism, can be employed to prevent such failures. However, utilizing such a mechanism is outside of the scope of this research.

B. SYSTEM LOGS AND LOG PARSING

System logs contain live information regarding the system status and events. These events can occur in the operating system, applications, or hardware devices. Logs include system events and statuses such as hardware changes, application events, errors, warnings, performance metrics, and security events. Logs help us understand the system state, debug performance issues, and perform root cause analysis.

The structure of each log file is different and has a unique format; however, each log line usually contains the timestamp at which it was logged, application information, and the actual log message. The log messages are text produced by logging statements in program code such as `printf()` and `logger.log()` [16]. The events are logged as they happen from currently running processes, so the logs are an ideal source of information for live system status investigation.

Unstructured log data must be parsed to make it structured and efficient for analysis [17]. An effective method is to extract the log message from each entry. We can assign a unique identifier for each message type called the log key. For this, the parser must distinguish between each raw log message's constant and variable parts. The constant part is analogous to the logging statement in the processes' source code. For example, the log template for log message, `"br0: port 2(tap1) entered forwarding state"` is `"<*>: port <*>(tap<*>) entered <*> state"` which is the string constant from some logging statement similar to,

```
printf("%s: port %d(tap%d) entered %s state",
↪  deviceName, portId, tapId, state)
```

Furthermore, Drain [16] is the state-of-the-art online log parser that runs in an unsupervised manner and utilizes a fixed-depth parsing tree to categorize incoming log messages.

C. LOG KEY SUBSEQUENCES

The parsed log file is a time series of the log keys. Given a log key time series $T = \{t_1, t_2, \ldots, t_{n-1}, t_n\}$, the subsequence of interval of length m to 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 by using any vector distance measures. The most popular and the one used in Matrix Profile [18] is the z-normalized Euclidean distance. For the given two subsequences $T_{i,m}$ and $T_{j,m}$, z-normalized Euclidean distance is defined as,

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

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

Given a subsequence $T_{i,m}$, the vector of distances between $T_{i,m}$ and each subsequence of T is the distance profile, $D_i = \{d_{i,m}, d_{i,m+1}, \ldots, d_{i,n-1}, d_{i,n}\}$, 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, that is, $P = \{min(D_m), min(D_{m+1}), \ldots, min(D_{n-1}), min(D_n)\}$ A small value in the Matrix Profile suggests that the sub-sequence pattern appears in other parts of the time series, known as a motif. In contrast, an unusually high Matrix Profile value indicates that the given sub-sequence is unique in the time series and could potentially represent an anomaly.

D. ANOMALY DETECTION USING MATRIX PROFILE

Given a Matrix Profile P for a time series T, anomaly detection becomes trivial because the anomalous subsequences have a high distance value compared to the other subsequences [18]. Using an appropriate threshold, we can effortlessly identify the anomalous subsequences in the time series, as shown in Fig. 1.

If we consider an online scenario, where the subsequences arrive one after the other, the future subsequences, that is the subsequences to the right-hand side of t_i (current time-step) are not known. The left matrix profile considers this

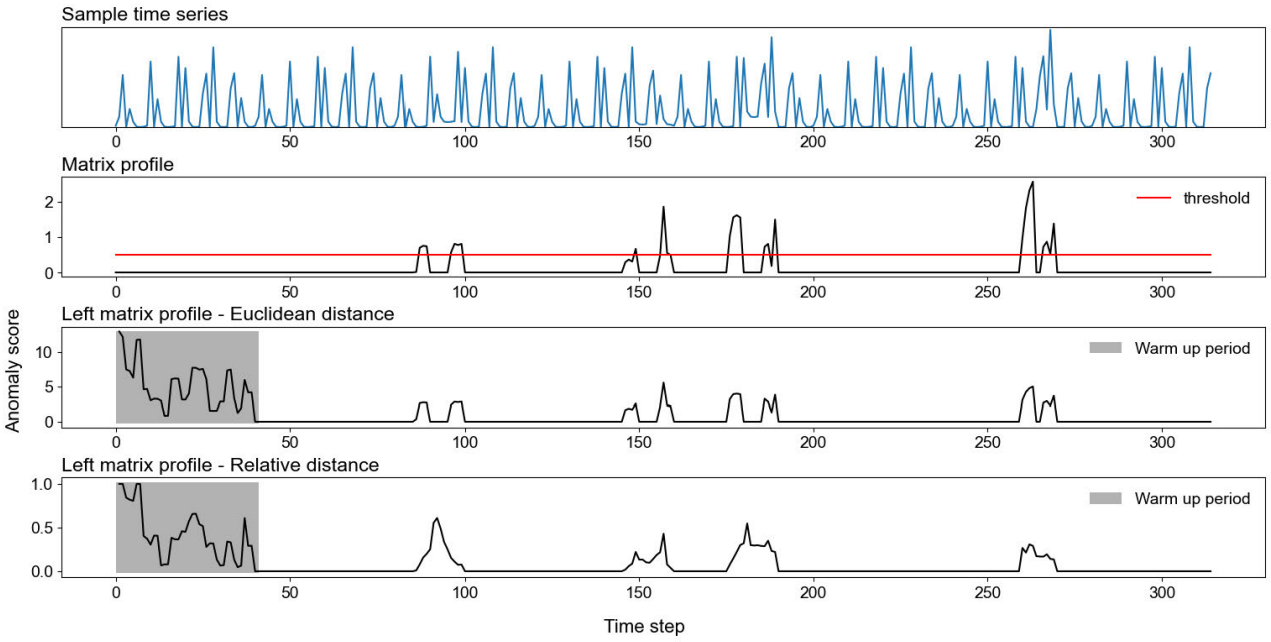


FIGURE 1. An example of a matrix profile and left matrix profiles using Euclidean distance and relative distance.

and calculates the distances for a subsequence at time step i based only on the left-hand-side subsequences. As shown in Fig. 1, the first few values of the left matrix profile are high because it does not have many subsequences to compare to. This is called the warm-up period of the left matrix profile [19].

We can take the left matrix profile value at time step i as the anomaly score for the data point i. However, the Euclidean distance (refer equation (1)) has no upper bound, which makes it challenging to obtain an appropriate anomaly threshold. As mentioned in [19], we can swap 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 [20], the relative distance measure would be ideal in this scenario, as it outputs a value between 0 and 1. The relative distance D^R is defined as,

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

Fig.1 shows the original time series, matrix profile, and left matrix profiles for Euclidean and relative distance measures for the same time series.

E. ANOMALY DETECTION WITH MACHINE LEARNING

Many researchers [11], [12], [17], [21], [22], [23], [24], [25] have proposed several real-time ML-based anomaly detection models in the past decade. While supervised models could have superior performance due to being trained on labeled data [11], [23], [24], [25], [26], they may be infeasible to utilize in dynamic environments where the drift in data leads to excessive re-labeling efforts, making these approaches often impractical.

Deep Learning (DL) based unsupervised/semi-supervised models like Autoencoders [22], [27], and LSTM [17], [28], [29] rely on learning benign patterns from a set of benign logs. While not needing to specifically understand all possible anomaly situations, there is still a need for an offline training phase to effectively learn benign patterns, which could be costly [26].

State-of-the-art semi-supervised log anomaly detection models like DeepLog [17] and PLELog [29] are designed to adapt to changing log patterns, either using human feedback (as in DeepLog) or using word embedding [9], [28], [29] to extract semantic information from log messages to estimate the label resulting in a more robust and superior anomaly detection capability. More recent log anomaly detection models [8], [30] utilize a transformer architecture [31] to extract more contextual information for even more robust log classification.

The need for such advanced models is driven by real-world scenarios where systems generate hundreds to thousands of data streams, making thorough labeling infeasible. Moreover, it's impossible to identify all potential anomalous situations a priori for training supervised models. These challenges emphasize the necessity for real-time models such as Hierarchical Temporal Memory (HTM) [12] and ARTime [32] which are able to learn benign patterns and train their internal weights in real-time, thereby being able to better adapt to changes in dynamic environments where unforeseen anomaly types may appear in the future.

F. NATURAL LANGUAGE UNDERSTANDING

All logging statements generated by software applications are written in natural language [9], [17], predominantly in English. This allows us to leverage natural language understanding (NLU) techniques to extract insights from the log data, rather than interpreting them at a superficial level. NLU, a field within Natural Language Processing (NLP), aims to enable machines to interact with human language and understand the meaning behind sentences. In recent years, we have seen a massive boom in this area, giving rise to transformer-based [31] large language models (LLMs) like BERT [33] and GPT [34]. Some of these models show comprehension ability that surpass human experts in specific domains [35], [36]. It is shown that the size of the LLM (number of parameters) is directly proportional to its performance [37]; however, there are some relatively small models fine-tuned for specific tasks that outperform large models [38]. 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 [39].

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.

III. DATA COLLECTION

This section discusses how log data is collected for testing and model evaluation. We collected log data from 4 physical machines over 5 months (nearly 170 days), 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 log data collection. The physical machines consisted of two IBM System x3560 M4 - 48-core Intel Xeon E5-2697v2 machines with 338 GB of memory connected with Gigabit Ethernet acting as a source and a monitoring server, and two HP Z620 Workstation - 12-core Intel Xeon E3-1200v3 machines with 16 GB of memory, also connected with Gigabit Ethernet acting as a source and a monitoring server.

While public log datasets, such as Loghub [40], are readily available, they have significant limitations for our specific research needs. They lack log datasets related to VM/Server failures, which is essential for our study. Moreover, they do not include critical features we require for analyzing the feasibility of proactive VM fault tolerance. Specifically, they are missing essential metadata, such as timestamps associated with the initial fault (in our case, the fault injection time) and the eventual failure. These timestamps are necessary for assessing the potential for implementing proactive fault tolerance mechanisms in VMs.

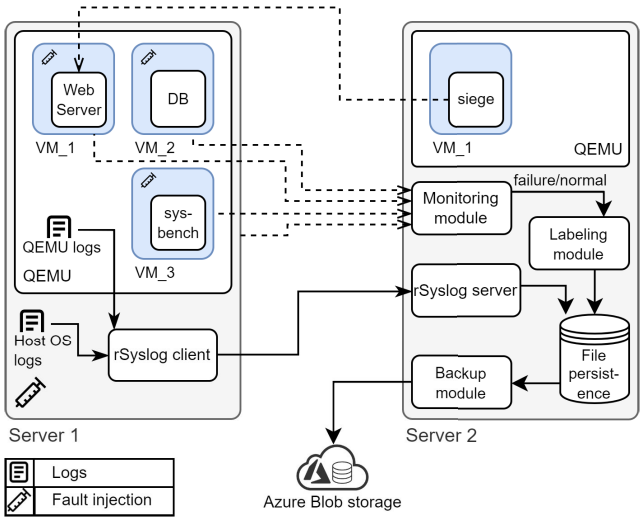


FIGURE 2. Data collection testbed architecture.

A. TESTBED

The high-level architecture diagram of our testbed setup is presented in the Fig. 2. 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.

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 [41] 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 [42] 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 GET request using a CRON (the periodic job scheduler of Linux) task to indicate that they are alive. HTTP requests not only verify the VM reachability but also verify that the applications running in VMs function properly.

In normal conditions and under heavy load, the VMs are expected to perform without any service disruption, without dropping any connection, so for a general scenario, we define the failure as the point at which the monitoring module does not receive a heartbeat pulse from the host or the VMs within 18 seconds. We chose 18 seconds because it gives sufficient time (3 extra seconds) to account for slight timing differences and other delays and to confirm total VM failure. For specific

scenarios, such as Out of Memory (OOM), we define the failure point differently, as described below (section III-B).

B. FAILURE SCENARIOS

According to Vishwanath and Nagappan [43], 78% of server failures are attributed to hard-disk-related issues, while 5% are caused by memory-module-related problems. According to Cano et al. [44], about 30% of the failures in private cloud servers are related to HDD failures, while 16% are caused by memory-related issues. The CPU and motherboard are considered the most reliable hardware components in servers [43], 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 [11], [45], such as physical machines running out of memory and very high CPU utilization. Handling of VM failures that originate from the network failure is outside the scope of this research because the network failure may impact the migration process, rendering VM migration impractical as a solution for network-related failures. Given the low probability of individual servers failing within a year and the need to study a variety of failure scenarios, 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 induced by over-allocating the total memory for VMs by 25% of the host's total physical memory capacity. In a cloud environment, OOM failure can occur if the server consolidation algorithm decides to allocate VMs over the available physical memory capacity of the host, due to host resource under-utilization by VMs, and if there is a sudden increase of VM memory utilization. Additionally, it can occur due to software faults or malicious software installed on the host. 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* [46] tool. We consider the VM failure as the log timestamp where the host OS invokes the OOM-killer to kill the VM process or the failure label by the monitoring module, whichever is earlier. Following are some sample logs collected for OOM failure,
```
qemu-system-x86 invoked oom-killer:
↪  gfp_mask=0x100cca(GFP_HIGHUSER_MOVABLE),
↪  order=0, oom_score_adj=0
Call Trace:
dump_stack+0x6d/0x8b
dump_header+0x4f/0x1eb
oom_kill_process.cold+0xb/0x10
out_of_memory+0x1cf/0x500
...
```
2) **Hard disk failure** - We simulated hard disk failure according to [7] 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. We ignore the label of the monitoring module as this is a simulated failure. Following are some sample logs collected for HDD failure,
```
blk_update_request: critical medium error,
↪  dev sdb, sector 4576 op 0x0:(READ) flags
↪  0x80700 phys_seg 32 prio class 0
FAILED Result: hostbyte=DID_OK
↪  driverbyte=DRIVER_SENSE
Sense Key : Medium Error [current]
Add. Sense: Unrecovered read error
CDB: Read(10) 28 00 00 00 12 30 00 00 08 00
...
```
3) **Buffer-IO error** - Buffer I/O 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 corruption [7]. Similar to hard disk failure, for buffer I/O errors, we defined failure as the point where we continuously encountered several errors, and as this is a simulated error, we ignore the label of the monitoring module. Following are some sample logs collected for Buffer-IO error,
```
buffer_io_error: 62 callbacks suppressed
Buffer I/O error on dev dm-0, logical block
↪  0, async page read
Buffer I/O error on dev dm-0, logical block
↪  1, async page read
...
```
4) **CPU over-utilization** - We induced VM failure due to CPU over-utilization by over-allocating the total VM vCPUs by 30% of the available host CPU cores. This failure can happen in a cloud environment if the server consolidation algorithm over-allocates VMs on the host, or due to software faults or malicious software installed on the host, which may lead to CPU overheating. During normal operation, the VMs functioned without any noticeable issues to performance. We used the *stress* [46] tool to simulate a failure by stressing the CPU on VMs and the host. For this failure, we consider the failure label from the monitoring module. Following are some sample logs collected for CPU over-utilization failure,
```
perf: interrupt took too long (3975 > 3970),
↪  lowering
↪  kernel.perf_event_max_sample_rate to
↪  50250
INFO: task qemu-system-x86:1431 blocked for
↪  more than 3 seconds.
Not tainted 5.4.0-166-generic #183-Ubuntu
qemu-system-x86 D    0  1431   1067
↪  0x00000000
Call Trace:
__schedule+0x2e3/0x740
schedule+0x42/0xb0
io_schedule+0x16/0x40
wait_on_page_bit+0x120/0x240
```

C. 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,

TABLE 1. Number of log lines in each dataset under each region.

Dataset	Log lines			
	Benign	Pre-failure	Post-failure	Total
HDD	79345	74574	10901	164820
OOM	96833	7365	26196	130394
Buffer-IO	369499	97918	16274	483691
CPU	58192	50187	13429	121808
Benign	113313	-	-	113313
			Total data points	1014026

as mentioned in Section II-B 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. Next, we utilized the DRAIN-3 [47] log parser, as explained in Section II-B above, to extract the log template, log template ID (log key) and parameters (values).

We simulated approximately 130-150 instances of each failure scenario, resulting in a total of *691 datasets* with more than *one million log lines*. Each dataset has 3 distinct regions: benign region (normal state before fault injection), pre-failure region (after fault injection), and post-failure region (after failure point). The datasets were formatted similar to Numenta Anomaly Benchmark (NAB) [48]. Table 1 presents the number of log lines in each dataset under each region of the dataset. Collected datasets and labels are available publicly.[1]

IV. 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. 3 illustrates the architecture of our model. It has three main sub-components: an anomaly detection module, a subsequence store, and an adaptive learning module to handle anomalous situations.

Our model runs time-stepped, each time-step is defined as the moment we receive the next log line, so receiving a new log line would be a new time step. 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 [18]. 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 [17], [20] utilize 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.

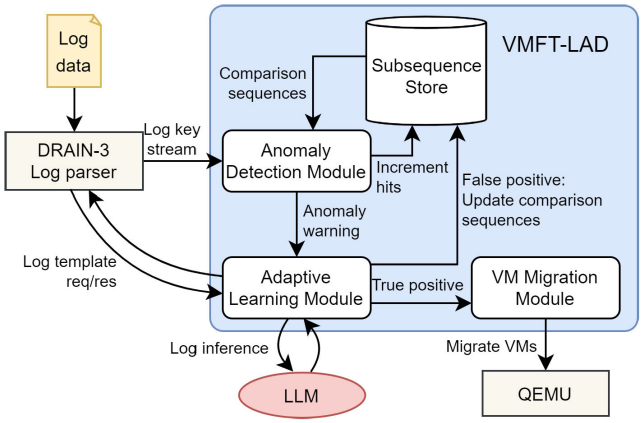


FIGURE 3. The architecture of VMFT-LAD.

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.

A. 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) [48].

1) The vanilla Matrix Profile was modified similar to [20] using the relative distance measure as discussed in Section II-D above instead of the z-normalized Euclidean distance.

[1]VM failure log dataset: https://github.com/CloudnetUCSC/NAB/tree/master/data

TABLE 2. Symbol legend for algorithms.

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

2) Unlike [20], we only utilize unique subsequences in the initial training period for comparison, dramatically reducing the model's memory footprint. We additionally 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 vanilla Matrix Profile [18], 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 avoid giving false positives to similar subsequences in the future.

Table 2 summarizes the symbols used in the algorithms.

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
```

Algorithm 1 presents our proposing 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 (discussed in the section IV-C) 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 II-D 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 utilize two different thresholds: the similarity threshold (η) and the anomaly threshold (θ). The similarity threshold (η) defines the distance at which we consider two subsequences similar, while the anomaly threshold (θ) determines the minimum distance at which we classify a subsequence as anomalous if no similar subsequence is found in the comparison set. The similarity threshold must be strictly less than the anomaly threshold ($\eta < \theta$); otherwise, the model would identify an anomalous subsequence as similar to some subsequence in the comparison set, resulting in no anomalies being detected.

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 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).

B. 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 a 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. Additionally, 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 [37], [39], 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 VII 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 25 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 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. One issue with directly using LLM output is that LLMs tend to have hallucination issues, that is, the text they generate might be incorrect, or nonsensical. However, here we incorporate only a binary result (either faulty or not), with minimal text generation (4 or 5 words), additionally, based on our LLM ablation study presented in section VII-C, while not perfect, the LLM output can differentiate between a failure-related anomalous log and a normal log line that was identified as anomalous but does not contain any failure-related information.

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 (normal-state) 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 in models like RAMP [20], where they mark such logs as anomalous with high confidence.

C. 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.

V. 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

A. EVALUATION CRITERIA

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

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.

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, in all the models except DeepLog, because DeepLog is pre-trained with benign data, whereas the other models including VMFT-LAD learn online during the learning period.

B. 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) [48], and DeepLog [17] 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 a 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: These are high-scoring LLMs based on the Hugging Face Open LLM Leaderboard [49][2]
- Bart Large (Zero-Shot): Bart Large is the most popular open-source zero-shot text classifier in the Hugging Face model library [2]

3) HTM: This model uses a different anomaly detection approach based on the Hierarchical Temporal Memory (HTM) architecture [12]. 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) [21] is a K-Nearest Neighbors (KNN) based non-parametric anomaly detection model.
5) EXPOSE: EXPected Similarity Estimation (EXPoSE) [50] is a non-parametric anomaly detection model based on a kernel function to measure similarity between data points.
6) ARTime: ARTime [32] is based on Adaptive Resonance Theory (ART), and this model outperforms the state-of-the-art model HTM in NAB.
7) DeepLog: DeepLog [17] is a popular log anomaly detection model based on an LSTM model; It uses Drain [16] as its log parser.

VMFT-LAD and other NAB models train online, so we define the first 150 data points of each dataset as the training region (which is roughly 15% of each dataset instance); we made sure that each instance of the dataset had at least 400 benign data points initially.

The chosen hyperparameters of the VMFT-LAD model for the evaluation are as follows - Window size (m): 5, learning period (M): 150, similarity threshold (η): 0.05, anomaly threshold (θ): varied from 0 to 1 for ROC. We also explored how these hyperparameters affect the performance of our model in the next section (section VI). For the NAB models, we use the default parameter values (some of the models were parameterless.)

All LLMs used to evaluate VMFT-LAD are conditioned with 25 randomly selected benign log samples so that the model can better understand 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.

The DeepLog model is pre-trained with 2400 benign log sequences from our collected dataset. We tuned the hyperparameters of the DeepLog log key anomaly detection model to obtain the best results, which are presented below. Hyperparameters - classes: 750, candidate keys: 250, window size: 20, No. of recurrent LSTM layers: 2, hidden layer size: 64

[2]As of February 2024.

C. EVALUATING MODEL PERFORMANCE: ROC CURVE ANALYSIS

Figure 4 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.

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 perform exceptionally well (AUC: 0.992-0.996), closely followed by the VMFT-LAD without feedback (AUC: 0.992), HTM (AUC: 0.988), and DeepLog (AUC: 0.95). 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 0.9), except for KNN-CAD (AUC: 0.659) and EXPOSE (AUC: 0.512), which showed relatively poorer performance.

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), the ARTime model (AUC: 0.780), and the DeepLog model (AUC: 0.783) 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) and, notably, the DeepLog model (AUC: 0.783) maintained their performance. The HTM model (AUC: 0.680) and VMFT-LAD with Emerton Monarch 7B (AUC: 0.669) LLMs perform reasonably well. Other models, including VMFT-LAD without feedback (AUC: 0.588) and especially ARTime (AUC: 0.458), struggled with this dataset.

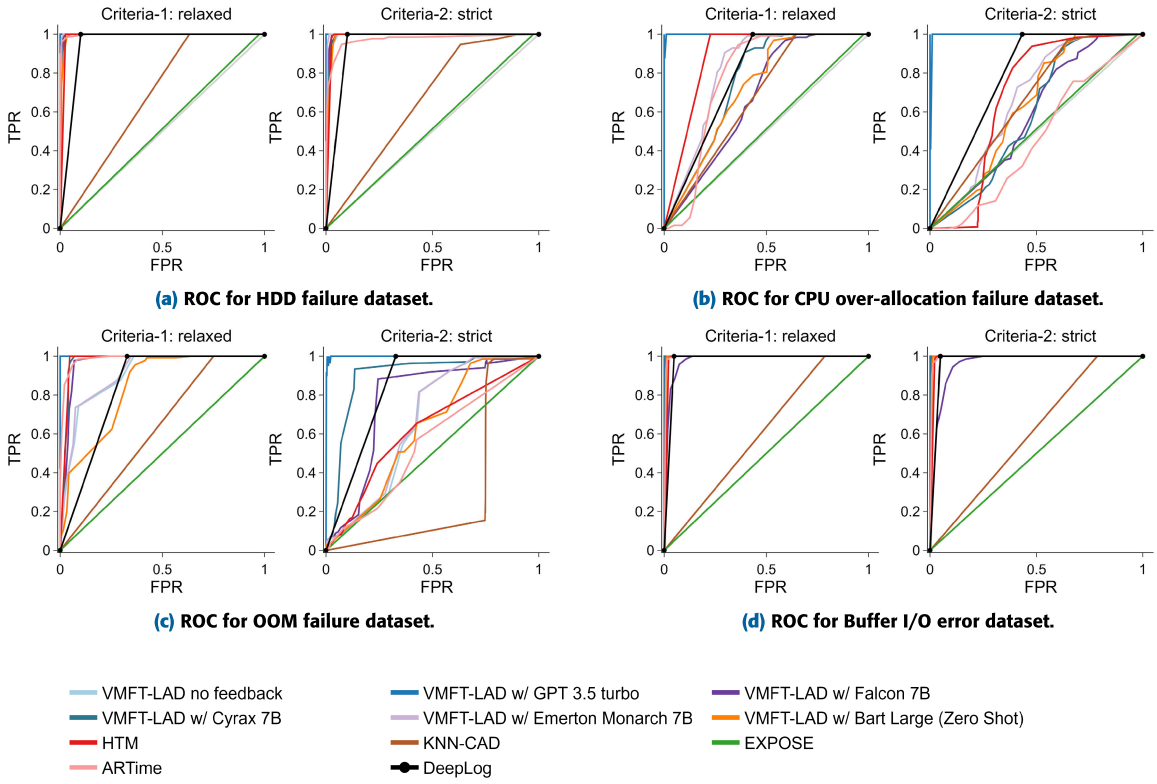


FIGURE 4. Receiver Operator Characteristics (ROC) curves for each dataset using all evaluated models.

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) and DeepLog also exhibit 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) and DeepLog (AUC: 0.836) models 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.

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. DeepLog model also performed well (AUC: 0.975). The VMFT-LAD with Falcon

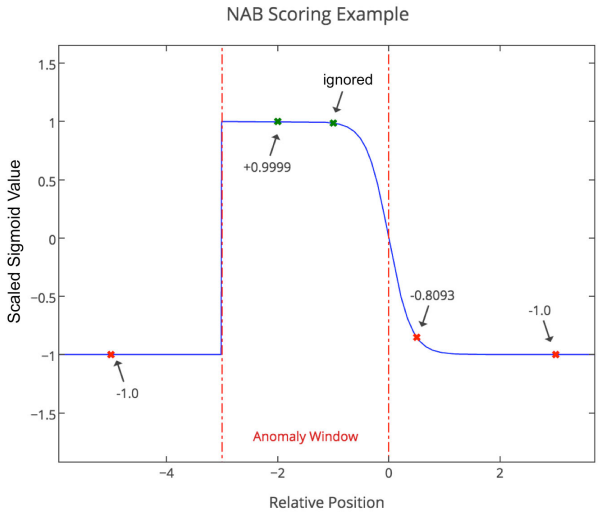


FIGURE 5. NAB sCORING example [48].

TABLE 3. Area Under the Curve (AUC) results for ROC curves.

Models		Dataset			
		HDD	OOM	Buffer-IO	CPU
Criteria-1: relaxed (Evaluates anomaly detection capability)	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
	DeepLog	0.95	0.836	0.975	0.783
Criteria-2: Strict (Evaluates early failure prediction capability)	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
	DeepLog	0.95	0.836	0.975	0.783

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 DeepLog model showed very good performance across the board. The VMFT-LAD without feedback and with other LLMs, such as Cyrax 7B and Emerton Monarch 7B, exhibit promising performance. The HTM and ARTime models perform well in certain scenarios, while KNN-CAD and EXPOSE generally struggle across the datasets and evaluation criteria.

D. EVALUATING MODEL PERFORMANCE: NUMENTA ANOMALY BENCHMARK

The Numenta Anomaly Benchmark (NAB) [48] 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. The NAB score is suitable for evaluating the models in this case because NAB gives a high score for early true anomaly detection. Additionally, it penalizes late predictions and false positives with negative marks using a sigmoidal scoring function [48]. Fig. 5 shows how the NAB scoring function scores the predictions of a model (marked as a cross) relative to their position to the anomaly window. The first true prediction within the anomaly window (green cross) gets a positive score, while all the other false positives (red crosses) get negative scores according to their closeness to the anomaly window.

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 anomaly window is defined as the pre-failure region (after fault injection) and the failure region (after the failure point) for the relaxed criterion, and for the strict criterion, we only consider the pre-failure region as the anomalous window and removed the post-failure region from the result set to allow NAB to score only the early pre-failure detections. The modified NAB repository is available publicly.[3]

Table 4 presents NAB score results under both Criteria-1: Relaxed and Criteria-2: Strict.

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), 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: 73.98), followed by

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

TABLE 4. NAB scores for evaluated models.

	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
	DeepLog	71.82	43.75	76.06
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
	DeepLog	71.50	43.30	75.79

the DeepLog model (standard score: 71.82), and the HTM model (standard score: 66.63). 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 DeepLog model maintains its performance with a standard score of 71.5. The other models, however, exhibit a significant drop in performance under this criterion, with very low scores across all three metrics.

TABLE 5. Average false positive rate and early detection rate at the best threshold.

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%
DeepLog	0.37%	100%

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 the 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 a high early failure indicator detection rate (96.28%). Notably, the DeepLog model shows a 100% early detection rate; however, its false positive rate is relatively high at 0.37%.

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.

E. MODEL EXECUTION TIME

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.

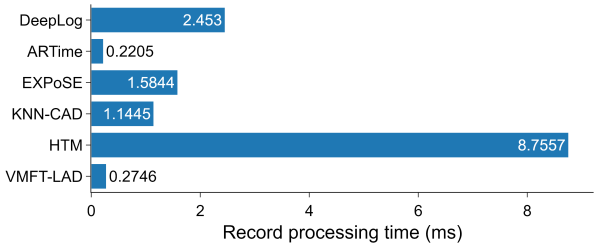


FIGURE 6. Average execution time to process a record (Lower the better).

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

The 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.

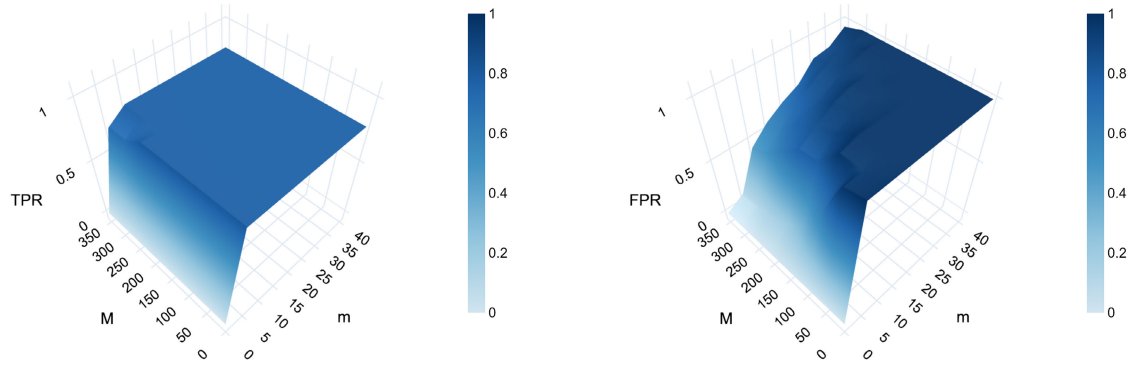
All the evaluated models demonstrate sufficient computational efficiency to handle online anomaly detection in our server environment. However, for scenarios with exceptionally high log generation rates, the VMFT-LAD model without LLM feedback and ARTime can be the most suitable choice due to their sub-millisecond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

VI. HYPERPARAMETER TUNING

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

Fig. 7a 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. 7b, 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,



(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.

allowing the model to identify anomalies within the time series data effectively.

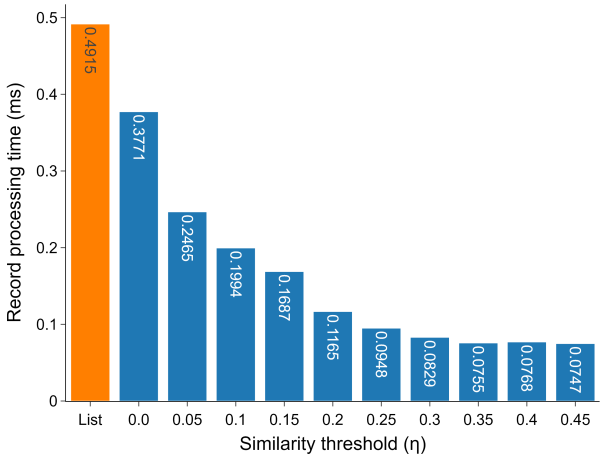


FIGURE 8. Impact of the similarity threshold (η) over the average record processing time. The average record processing time using a list-based implementation is added for comparison.

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 IV-A 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.

η does not affect the TPR or the FPR of the model because they are determined by the anomaly threshold θ. Additionally, we 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 according to 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 most evaluated models.

VII. DISCUSSION

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

A. 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 log anomaly detection time of models in comparison to the failure time of the VMs under each fault model, to assess their ability in detecting failure in advance to allow sufficient time for VM migration.

Table 6 presents the average early detection time before the VM failure for each model across all our datasets. 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 log dataset for OOM failures.

TABLE 6. Average detection time before total VM failure (minutes).

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
DeepLog	15.675	4.376	13.354	23.959

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 [51]. We ran Memcached 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.

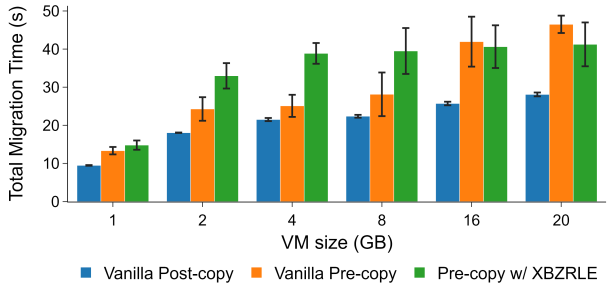


FIGURE 9. The total migration time for migrating VMs with different v-RAM sizes.

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 all the models 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

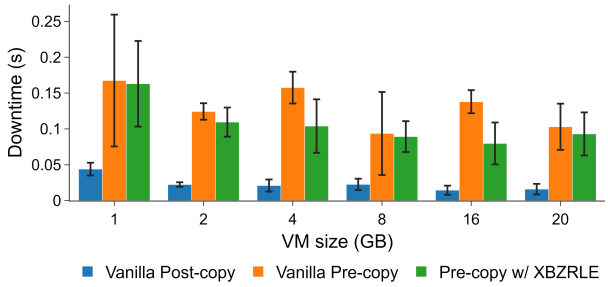


FIGURE 10. The downtime for migrating VMs with different v-RAM sizes.

achieves an average detection time of 3.033 minutes, the migration time for a 20 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.

These results, combined with the previous evaluation 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.

B. EXPLORING LLM USAGE PARADIGMS

VMFT-LAD reduces false positives and continuously adapts to changing logging patterns by integrating LLM feedback, eliminating the need for human intervention in the VM failure prediction process. For the LLM feedback, we have 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 as explained below (section VII-C). Additionally, we attempted to fine-tune the distilBert [38] LLM using Low-Rank Adaptation (LoRA) [52], 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.

C. COMPARING LLM PERFORMANCE

In this section, we compare the performance of LLMs used in the evaluation of our model VMFT-LAD using AUC results and the average false positive rates.

Table 7 presents the AUC results for the LLMs on each dataset. Most of the models show relatively low AUC results, around 0.5, except for the GPT 3.5 turbo model, which demonstrates a near-perfect AUC score (0.999) for all datasets, with the exception of the OOM dataset (AUC: 0.87).

Table 8 presents the False Positive Rate of the LLMs on our datasets. The Falcon 7B, Cyrax 7B, and the Emerton Monarch 7B models have similar FPR around 7-12%. The Bart Large zero-shot classifier had the worst FPR at 17.49%.

TABLE 7. Area Under the Curve (AUC) results for ROC curves of LLMs.

	Models	Dataset			
		HDD	OOM	Buffer-IO	CPU
Criteria-1	Falcon 7B	0.517	0.499	0.505	0.534
	Cyrax 7B	0.506	0.5	0.501	0.515
	Em. Monarch 7B	0.518	0.499	0.502	0.515
	Bart Large (0-Shot)	0.513	0.499	0.501	0.524
	GPT 3.5 turbo	0.999	0.87	0.999	0.999
Criteria-2	Falcon 7B	0.517	0.499	0.487	0.534
	Cyrax 7B	0.506	0.5	0.501	0.515
	Em. Monarch 7B	0.518	0.499	0.502	0.515
	Bart Large (0-Shot)	0.513	0.499	0.501	0.524
	GPT 3.5 turbo	0.999	0.87	0.999	0.999

TABLE 8. Average false positive rate for LLMs.

Model	False positive rate
Falcon 7B	7.95%
Cyrax 7B	8.78%
Emerton Monarch 7B	12.07%
Bart Large (Zero-Shot)	17.49%
GPT 3.5 turbo	0.05%

Impressively, the GPT 3.5 turbo model showed a very low FPR of 0.05%. These results show that using few-shot learning is much better for differentiating between a true VM failure-indicating log and a normal but anomalous log.

The GPT 3.5 turbo model showed good AUC performance and low FPR; when comparing this to using VMFT-LAD in conjunction with the GPT 3.5 turbo model, we can see that VMFT-LAD had an improvement over the results of only using GPT 3.5 turbo model; specifically in the case of the OOM dataset and the FPR (combined model FPR: 0.02%)

The promising results for the GPT 3.5 turbo model raises an intriguing possibility: could an LLM like GPT 3.5 turbo, given adequate resources, be solely used for the VM failure prediction task through log analysis? While the results are promising, further research and analysis are necessary to draw definitive conclusions. A significant challenge to this approach lies in the latency issue. The time interval between logs (163.9 ms) is substantially shorter than the LLM's response time (approximately 800 ms - 2 seconds), making it impractical to query the LLM for each individual log line. These results show how VMFT-LAD and an LLM like GPT 3.5 turbo complement each other in creating a highly effective and efficient log anomaly detector for VM failure prediction. A practical implementation could involve installing VMFT-LAD instances on each physical host in the data center, with a single LLM instance serving requests for anomalous log classification (differentiating between normal and failure-related logs) from all deployed VMFT-LAD instances across the hosts. This setup would balance the strengths of both models while mitigating the latency issues.

D. PROPERTIES OF AN IDEAL VM FAILURE PREDICTOR

In this section, we show how our model aligns with the properties of an ideal VM failure predictor, as outlined in the introduction section.

The VMFT-LAD model exhibits several key characteristics that make it an effective VM failure predictor:

1) *Early identification of failures:* As demonstrated in the evaluation section above, our model successfully identifies failures at an early stage while maintaining a very low false positive rate.
2) *Adaptability to changing environments:* When the log pattern changes in the host machine due to software or hardware update, the previously learned "normal" state of any anomaly detector must be updated; otherwise the new normal state would be marked as an anomaly. When VMFT-LAD identifies an abnormal log sequence, it will consult an LLM to verify whether it is a true anomaly (VM failure indicator) or normal log unrelated to VM failure. If the anomaly is a normal log, VMFT-LAD updates its internal state to adapt to the new change.
3) *Ability to identify unforeseen failure types:* By training only on the normal system state logs and identifying anomalies, VMFT-LAD is inherently designed to detect unforeseen failures, as all failure indicators are anomalies.
4) *Capability to work with highly imbalanced data:* VM failure data are highly imbalanced as an individual server will be in the normal operational condition most of the time, so the system will only have normal state log data and failures are relatively very rare. Semi-supervised models like VMFT-LAD can handle highly imbalanced data, as they only train on one class (normal class).
5) *Minimization of false positives:* VMFT-LAD effectively minimizes false positives with the help of LLM feedback.
6) *Ability to work independently:* Once deployed, VMFT-LAD operates autonomously, requiring minimal human intervention in identifying VM failure-indicating logs. This autonomy results in faster reaction times when managing VM failure scenarios.

These properties demonstrate how VMFT-LAD functions as a robust and efficient VM failure prediction system, addressing key challenges in proactive fault tolerance.

VIII. RELATED WORK

This section reviews related literature on VM proactive fault tolerance using machine learning techniques to contextualize our contributions and highlight the research gaps that our work addresses.

A. PHYSICAL MACHINE FAILURE PREDICTION

Physical machine (server) failure prediction is an integral part of VM fault tolerance because the physical machine

failure will inevitably cause the VMs running on it to fail. In large-scale cloud computing environments, physical machine failures are inevitable, with studies indicating that 6-8% of servers experience at least one hardware issue annually [43], [45]. Researchers have extensively explored server failure prediction by leveraging historical resource usage data and ML techniques. The common approach involves training supervised ML models, such as random forests [53], convolutional neural networks (CNNs) [10], and bi-directional long short-term memory (LSTM) [54] models, on labeled data obtained from system administrators.

One notable framework is MING [13], developed by Microsoft Research, which employs LSTM and random forest models on temporal data (performance, log rates, OS events) and spatial data (server rack location, load-balancer data), respectively, to predict server failures. MING also incorporates a server ranking system to identify the most failure-prone servers.

B. VIRTUAL MACHINE FAILURE PREDICTION

VM failures can be due to underlying physical server failures or issues with software components like the host OS, hypervisor, or guest OS. Traditional approaches to VM fault tolerance involve employing redundant VMs [55], which can be costly for cloud service providers. An alternative strategy is to predict VM failures and proactively migrate failure-prone VMs to other physical servers.

Similar to physical machine failure prediction, several studies have focused on using supervised ML models, such as LSTM, CNN, linear regression, support vector machines (SVMs), and feed-forward neural networks (FNNs), on VM resource usage data to identify failure-prone VMs for proactive migration [11].

However, research on VM failure prediction using log analysis remains relatively scarce. Most existing studies [10], [11], [23] have primarily focused on utilizing physical machine resource usage history, overlooking the potential insights provided by VM and server logs. These logs contain valuable information about VM behavior, performance, and potential failure indicators, which can significantly enhance failure prediction accuracy.

The framework proposed by Nam et al. [9] focuses on predicting failures of Virtual Network Functions (VNFs) by leveraging log data generated by the VNF application and the VM. They employ word embedding using Google *Word2Vec* [56] and a supervised CNN model for failure prediction. In a subsequent work [8], the authors improved their approach by utilizing the BERT tokenizer [33] for word embedding and a CNN model for prediction. However, these studies have been limited to specific types of VMs, such as VNFs, and have not been applied to generic VMs commonly used in cloud computing environments.

Most VM/server failure prediction frameworks [8], [9], [10], [11], [23] discussed above employ supervised ML models, which require human effort or other methods to label the data as normal or failure (pre-failure) classes, and this approach is inefficient in large-scale systems such as CDCs. Moreover, once such a model is deployed, it may not function properly if the system experiences even slight changes or faces unseen failure types, rendering it ineffective in handling unforeseen failures. In real-world scenarios, failure data are rare, making it impractical and infeasible to identify all failures a priori to train a perfect supervised model. Unsupervised or semi-supervised models are preferred, as they can train only on the readily available normal state of the system.

Additionally, an important factor that has been largely disregarded in previous studies is the consideration of the time required for VM migration. For successful VM migration, it is imperative to identify the failure before the total time it takes to migrate the VM to a healthy physical machine. Unfortunately, most studies have neglected to evaluate this aspect of their work.

As demonstrated by our evaluation results, our work addresses these research gaps by providing a comprehensive approach to fault tolerance of generic VMs through log analysis. Our semi-supervised approach eliminates the need for labeled data, while effectively identifying previously unseen VM failure situations in real-time and adapting to the changes in normal log patterns of the system. We have also considered the VM migration timing requirements when evaluating our approach.

IX. CONCLUSION

This paper presents VMFT-LAD, a semi-supervised log anomaly detection model designed for proactive VM fault tolerance. By combining the efficiency of our modified Matrix Profile [18] with the log inference capability of large language models (LLMs), VMFT-LAD addresses the limitations of traditional approaches and enables early detection of potential failures, including unforeseen fault types, while continuously adapting to changing log patterns with minimal human intervention. Our comprehensive evaluation of VMFT-LAD on several datasets exemplifies its superiority over state-of-the-art real-time anomaly detection models.

While this work primarily focuses on log data analysis, we are currently exploring future directions to further improve the model's capabilities. One promising direction we are investigating is the integration of both resource usage metrics and log data for more comprehensive VM failure prediction. This approach has the potential to significantly enhance the model's predictive capabilities.

In conclusion, VMFT-LAD represents a meaningful step forward in facilitating proactive VM fault tolerance using log analysis, offering early and accurate failure prediction in dynamic, complex cloud environments.

ACKNOWLEDGMENT

The authors would like to sincerely thank the School of Computing, University of Colombo, for providing the necessary infrastructure and WSO2 LLC for donating the

cluster-tested hardware to set up the test environment for this research work. The resources were invaluable in conducting the evaluations and experiments presented in this article.

REFERENCES

[1] R. Jhawar and V. Piuri, "Fault tolerance and resilience in cloud computing environments," in *Computer and Information Security Handbook*, 3rd ed., J. R. Vacca, Ed., Boston, MA, USA: Morgan Kaufmann, 2017, pp. 165–181. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128038437000090

[2] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *Proc. 17th Euromicro Int. Conf. Parallel, Distrib. Netw.-based Process.*, 2009, pp. 252–257.

[3] D. Fernando, J. Terner, K. Gopalan, and P. Yang, "Live migration ate my VM: Recovering a virtual machine after failure of post-copy live migration," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 343–351.

[4] D. Fernando, J. Terner, P. Yang, and K. Gopalan, "V-recover: Virtual machine recovery when live migration fails," *IEEE Trans. Cloud Comput.*, pp. 1–12, 2023.

[5] A. Polze, P. Tröger, and F. Salfner, "Timely virtual machine migration for pro-active fault tolerance," in *Proc. 14th IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput. Workshops*, Mar. 2011, pp. 234–243.

[6] R. Miller. (2008). [Online]. Available: https://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers

[7] 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 *Proc. 6th South-East Eur. Design Autom., Comput. Eng., Comput. Netw. Social Media Conf. (SEEDA-CECNSM)*, Sep. 2021, pp. 1–7.

[8] S. Nam, J.-H. Yoo, and J. W. Hong, "VM failure prediction with log analysis using BERT-CNN model," in *Proc. 18th Int. Conf. Netw. Service Manage. (CNSM)*, Oct. 2022, pp. 331–337.

[9] S. Nam, J. Hong, J.-H. Yoo, and J. W. Hong, "Virtual machine failure prediction using log analysis," in *Proc. 22nd Asia–Pacific Netw. Oper. Manage. Symp. (APNOMS)*, Sep. 2021, pp. 279–284.

[10] X. Sun, K. Chakrabarty, R. Huang, Y. Chen, B. Zhao, H. Cao, Y. Han, X. Liang, and L. Jiang, "System-level hardware failure prediction using deep learning," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1–6, doi: 10.1145/3316781.3317918.

[11] D. Saxena and A. K. Singh, "OFP-TM: An online VM failure prediction and tolerance model towards high availability of cloud computing environments," *J. Supercomput.*, vol. 78, no. 6, pp. 8003–8024, Apr. 2022.

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

[13] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang, "Predicting node failure in cloud service systems," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 480–490.

[14] R. Jhawar and V. Piuri, "Fault tolerance management in IaaS clouds," in *Proc. IEEE 1st AESS Eur. Conf. Satell. Telecommun. (ESTEL)*, Oct. 2012, pp. 1–6.

[15] D. Fernando, H. Bagdi, Y. Hu, P. Yang, K. Gopalan, C. Kamhoua, and K. Kwiat, "Quick eviction of virtual machines through proactive live snapshots," in *Proc. IEEE/ACM 9th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2016, pp. 99–107.

[16] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jun. 2017, pp. 33–40.

[17] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1285–1298.

[18] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, "Matrix profile I: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *Proc. IEEE 16th Int. Conf. Data Mining (ICDM)*, Dec. 2016, pp. 1317–1322.

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

[20] J. D. Herath, C. Bai, G. Yan, P. Yang, and S. Lu, "RAMP: Real-time anomaly detection in scientific workflows," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 1367–1374.

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

[22] H. Xu, Y. Feng, J. Chen, Z. Wang, H. Qiao, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, and D. Pei, "Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in web applications," in *Proc. World Wide Web Conf. World Wide Web (WWW)*, 2018, pp. 187–196.

[23] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proc. Internet Meas. Conf.*, Oct. 2015, pp. 211–224.

[24] 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," 2017, *arXiv:1708.03665*.

[25] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at Microsoft," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 3009–3017.

[26] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2016, pp. 207–218.

[27] A. Farzad and T. A. Gulliver, "Unsupervised log message anomaly detection," *ICT Exp.*, vol. 6, no. 3, pp. 229–237, Sep. 2020.

[28] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 4739–4745.

[29] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1448–1460.

[30] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2020, pp. 1196–1201.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.

[32] M. Hampton. (2021). *Artimenab*. [Online]. Available: https://github.com/markNZed/ARTimeNAB.jl

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

[34] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," Tech. Rep., 2018.

[35] X. Luo et al. (2024). *Large Language Models Surpass Human Experts in Predicting Neuroscience Results*. [Online]. Available: https://api.semanticscholar.org/CorpusID:268253470

[36] D. Van Veen, C. Van Uden, L. Blankemeier, J.-B. Delbrouck, A. Aali, C. Bluethgen, A. Pareek, M. Polacin, E. P. Reis, and A. Seehofnerova, "Clinical text summarization: Adapting large language models can outperform human experts," *Res. Square*, Oct. 2023.

[37] T. B. Brown, "Language models are few-shot learners," in *Proc. NIPS*, 2020, pp. 1877–1901.

[38] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[39] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 22199–22213.

[40] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for AI-driven log analytics," in *Proc. IEEE 34th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2023, pp. 355–366.

[41] J. Fulmer. *Siege—An Http Load Tester and Benchmarking Utility*. Accessed: Aug. 17, 2024. [Online]. Available: https://github.com/JoeDog/siege

[42] *Azure Blob Storage*. Accessed: Aug. 17, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/storage/blobs

[43] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 2010, pp. 193–204.

[44] I. Cano, S. Aiyar, and A. Krishnamurthy, "Characterizing private clouds: A large-scale empirical analysis of enterprise clusters," in *Proc. 7th ACM Symp. Cloud Comput.*, Oct. 2016, pp. 29–41.
[45] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, "Failure analysis of virtual and physical machines: Patterns, causes and characteristics," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 1–12.
[46] *Stress-Tool to Impose Load on and Stress Test Systems*. Accessed: Aug. 17, 2024. [Online]. Available: https://linux.die.net/man/1/stress
[47] IBM. (2020). *Drain3*. Accessed: Mar. 30, 2024. [Online]. Available: https://github.com/IBM/Drain3
[48] A. Lavin and S. Ahmad, "Evaluating real-time anomaly detection algorithms—The numenta anomaly benchmark," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2015, pp. 38–44.
[49] N. L. E. Beeching, and S. Han. (2023). *Open Llm Leaderboard*. [Online]. Available: https://huggingface.co/spaces/HuggingFaceH4/openllmleaderboard
[50] M. Schneider, W. Ertel, and F. Ramos, "Expected similarity estimation for large-scale batch and streaming anomaly detection," *Mach. Learn.*, vol. 105, no. 3, pp. 305–333, Dec. 2016.
[51] A. Shribman and B. Hudzia, "Pre-copy and post-copy vm live migration for memory intensive applications," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2012, pp. 539–547.
[52] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," 2021, *arXiv:2106.09685*.
[53] 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, Jan. 2012.
[54] J. Gao, H. Wang, and H. Shen, "Task failure prediction in cloud data centers using deep learning," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1411–1422, May 2022.
[55] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 4, pp. 30–39, Dec. 2010.
[56] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013, *arXiv:1310.4546*.

PRATHEEK SENEVIRATHNE received the B.Sc. degree in computer science from the School of Computing, University of Colombo, Sri Lanka. He is currently a Software Engineer with Pagero, Sri Lanka. His research interests include the applications of machine learning, virtualization, and fault-tolerance.

SAMINDU COORAY is currently pursuing the degree with the School of Computing, University of Colombo, Sri Lanka. He is a Research Intern with the Cloudnet Research Group, School of Computing, University of Colombo. His research interests include virtualization and networking.

JEROME DINAL HERATH received the Ph.D. degree from the Computer Science Department, State University of New York (SUNY) at Binghamton, USA. He is currently a Security Data Scientist with Obsidian Security, USA. His research interests include the applications of machine learning and deep learning, adversarial artificial intelligence (AI), and explainable AI.

DINUNI FERNANDO received the Ph.D. degree from the Computer Science Department, State University of New York (SUNY) at Binghamton, USA. She is currently a Senior Lecturer with the School of Computing, University of Colombo. Her research interests include virtualization, software-defined networks, and security.