Machine Learning for Anomalies Detection in Real-time Cloud

Mahsa Raeiszadeh

A Thesis
in
Concordia Institute
for
Information System Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy (Information and System Engineering) at
Concordia University
Montréal, Québec, Canada

February 2024

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mahsa Raeiszadeh**
Entitled: **Machine Learning for Anomalies Detection in Real-time Cloud**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and System Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.
Signed by the Final Examining Committee:

_____ Chair
Dr. Ferhat Khendek

_____ External Examiner
Dr. Foutse Khomh

_____ External to Program
Dr. Anjali Agarwal

_____ Examiner
Dr. Chadi Assi

_____ Examiner
Dr. Carol Fung

_____ Supervisor
Dr. Roch Glitho

Approved by _____
Dr. Farnoosh Naderkhani, Graduate Program Director

Date of Defence: March 24, 2025

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Machine Learning for Anomalies Detection in Real-time Cloud

Mahsa Raeiszadeh, Ph.D.
Concordia University, 2025

Cloud computing enables on-demand access to shared resources hosted in data centers and managed by cloud service providers. However, as cloud environments scale in size and complexity, they become increasingly prone to anomalies—deviations from expected behavior—that can disrupt reliability and availability. In real-time clouds, where operations must be completed within strict time constraints, anomalies pose a greater risk, potentially causing cascading failures, degraded performance, and increased maintenance costs. To address these issues, efficient methods for anomaly detection in real-time cloud environments are essential for maintaining service quality and operational efficiency.

Machine Learning (ML) has emerged as a promising approach for detecting anomalies in real-time clouds. By analyzing high-dimensional data such as system logs, traces, and performance metrics, ML models can identify complex patterns and deviations in dynamic, large-scale, and heterogeneous environments. However, employing ML in real-time clouds introduces several challenges, including handling sequential performance metrics, where evolving system behaviors cause concept drift, degrading model accuracy and requiring rapid adaptation to maintain low-latency anomaly detection; analyzing distributed traces, where inter-service dependencies and dynamic workloads introduce latency-sensitive bottlenecks, making timely anomaly detection difficult; and detecting anomalies in contextual logs, where log instability, class imbalance, and labeling dependency hinder model learning, further complicating real-time anomaly response under strict time constraints.

This thesis addresses these challenges with three key contributions for real-time cloud environments, where low-latency, adaptive, and scalable anomaly detection is critical. First, we propose a concept drift adaptation algorithm that integrates prediction-driven anomaly detection and adaptive window-based methods. This approach ensures effective handling of concept drift by dynamically adjusting to changes in the data distribution, enhancing detection accuracy over time. Second, we introduce a graph-based learning approach that captures inter-service dependencies while leveraging collaborative learning to reduce computational overhead and enable real-time updates. Third, we present a self-supervised log anomaly detection that adapts to evolving log structures without requiring labeled data, improving detection efficiency in dynamic cloud environments.

Acknowledgments

First and foremost, I express my deepest gratitude to my Ph.D. supervisor, Prof. Roch Glitho, for his invaluable guidance, time, and resources. His insights, encouragement, and mentorship have been instrumental in shaping my academic journey and making this Ph.D. experience both productive and rewarding.

I sincerely thank my Ph.D. committee members—Dr. Anjali Agarwal, Dr. Chadi Assi, and Dr. Carol Fung—for their constructive feedback, valuable suggestions, and unwavering support throughout this process. I am also grateful to the external examiner, Dr. Foutse Khomh, for his time and for accepting to serve on my Ph.D. thesis committee.

I extend my heartfelt appreciation to my collaborators at Ericsson, Dr. Johan Eker and Dr. Raquel Mini, for their inspiring discussions and for providing me with the opportunity to learn and grow through our collaborations. I am thankful to Dr. Amin Ebrahimzadeh and Dr. Felipe Estrada-Solano for their insightful feedback and support.

Special thanks go to my colleagues at the TSE lab for their encouragement. I am particularly grateful to Dr. Behshid Shayesteh for the enriching discussions and the much-needed moments of joy during lunch and coffee breaks.

I am deeply grateful to my mother, Mahboubeh, for always believing in me and encouraging me to follow my dreams. She played a big role in my decision to continue my study abroad, always pushing me to do my best and never give up. Her love, sacrifices, and constant support have shaped who I am today, and I will always be thankful to her. I also extend my gratitude to my father, Naser, for instilling in me the value of education and the drive to succeed. He is the most selfless and giving person I know, devoting his entire life to his children's happiness and success, sacrificing endlessly to ensure that we never lacked anything. To my sisters, Anahita and Marjan—you have been my greatest friends and unwavering cheerleaders, always lifting my spirits when I needed them most. More than just sisters, you have been my closest confidantes, my biggest supporters, and the ones I could always count on.

Last but not least, Amir, I cannot express enough how much your love, understanding, and support have meant to me through every high and low of this journey. You have been my rock and my greatest source of strength. Thank you for always believing in me, for lifting me up when I doubted myself, and for pushing me to keep going no matter what. I am endlessly grateful to have you by my side.

Contents

List of Figures

List of Tables

List of Abbreviations

AE	Autoencoder
AI	Artificial Intelligence
ATR	Automated Transport Robot
AUC	Area Under Curve
BERT	Bidirectional Encoder Representations from Transformers
BGL	Blue Gene/L
CNN	Convolutional Neural Network
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
DFLF	Dynamic Frequency-based Log Filtering
DL	Deep Learning
EO	Encoder-Only
ERDC	Ericsson Research Data Center
FL	Federated Learning
GA	Genetic Algorithm
GRU	Gated Recurrent Unit
HDFS	Hadoop Distributed File System
IIoT	Industrial Internet of Things
IoT	Internet of Things
IRCI	Infrequent-based Reconstruction with Complete Input

IRFI Infrequent-based Reconstruction with Frequent Input

KD Knowledge Distillation

LSTM Long Short-Term Memory

MES Manufacturing Execution System

ML Machine Learning

OCSVM One-Class Support Vector Machine

PCA Principal Component Analysis

PU Positive-Unlabeled

QoS Quality of Service

RCI Reconstruction with Complete Input

RNN Recurrent Neural Network

ROS Robot Operating System

SCG Service Call Graph

SLA Service Level Agreement

SLO Service Level Objective

SVM Support Vector Machine

VLSTM Variational Long Short-Term Memory

Chapter 1

Introduction

1.1 Overview

Cloud computing enables on-demand access to a shared pool of resources such as networks, servers, and storage. These resources, hosted in data centers managed by cloud providers, are delivered as services to users. However, as cloud environments grow in scale and complexity, they become prone to anomalies (i.e., data patterns that deviate from normal behavior) [8]. Anomalies can propagate through the cloud environment, causing service interruptions and violations of expected reliability and availability guarantees.

Ensuring reliability in cloud services is particularly challenging for real-time cloud environments, which prioritize deterministic, low-latency responses over general cloud goals like scalability and cost efficiency. Real-time clouds are designed to execute critical operations within strict time constraints (e.g., milliseconds), making them essential for latency-sensitive applications such as autonomous systems or smart manufacturing. To meet these stringent demands, real-time clouds must ensure high availability and reliability to trigger the correct actions within specified time frames.

The dynamic nature of streaming data in real-time clouds, which is heterogeneous and large-scale, demands real-time analytics within strict deadlines [9]. Moreover, real-time cloud environments heavily adopt native cloud architectures, particularly microservices, which enable scalability but also introduce new challenges for anomaly detection. Due to asynchronous communication, high inter-service dependencies, and dynamic workload variations, detecting anomalies in such environments is significantly more complex. Consequently, ensuring the availability and reliability of real-time cloud systems in the face of anomalous behaviors—such as unexpected hardware or software failures—is critical. Anomalies in real-time clouds can result in Service Level Agreement (SLA) violations, data loss, and sudden execution terminations, leading to degraded performance, reliability issues, and significant maintenance costs for service providers [10].

Traditional rule-based or manual anomaly detection approaches struggle to keep up with the speed, scale, and variability of real-time cloud environments. As a result,

automated anomaly detection with minimal human intervention has become essential. Machine Learning (ML), particularly Deep Learning (DL), provides a promising approach for enabling such automation. ML models can learn from diverse data sources, including logs, traces, and performance metrics (e.g., CPU, memory, disk usage), to detect anomalies.

Although ML is a promising approach, its deployment in the real-time cloud faces significant challenges. First, real-time clouds exhibit dynamic and heterogeneous system performance metrics, leading to high variability. This makes it difficult to establish consistent patterns for training ML models. Additionally, concept drift—where data distributions change over time—can degrade ML model performance [11]. To maintain accuracy, frequent retraining and adaptation are required. Second, the interconnected and distributed nature of real-time cloud services introduces additional complexity. These services generate hierarchical and contextual data from traces and logs, requiring anomaly detection systems to analyze intricate relationships across distributed components effectively. This is particularly critical in real-time clouds, where even minor delays in anomaly detection can have cascading effects. Third, the variation and evolution of log data formats in real-time clouds add another layer of difficulty. Logs, a primary source of information for anomaly detection, frequently change in format and structure as cloud systems evolve. In real-time clouds, where strict deadlines must be met, robust mechanisms are required to dynamically adapt to log format variations while maintaining effective anomaly detection. This thesis aims to develop robust ML-based techniques for anomaly detection in real-time cloud environments. By addressing key challenges such as data variability, service dependencies, and log evolution, this work contributes to enhancing the reliability and responsiveness of real-time cloud services. The following subsections discuss the specific challenges of employing ML for anomaly detection, outline the thesis contributions, provide background on key concepts, and present the thesis structure.

1.2 Challenges

This section outlines the key challenges associated with employing ML models for anomaly detection in real-time cloud environments.

Sequential Metrics in ML-based Anomaly Detection: ML-based anomaly detection in real-time cloud relies heavily on sequential performance metrics such as latency, throughput, and resource utilization. These metrics, which evolve dynamically over time, pose significant challenges for anomaly detection models due to their inherent temporal dependencies, high data velocity, and the need for low-latency decision-making. A key challenge is concept drift, which degrades model accuracy as data distributions shift over time. Additionally, the high frequency and large scale of data streams demand real-time processing to ensure timely detection. The presence of temporal dependencies and feature correlations further complicates anomaly detection compared to static datasets.

Distributed Traces in ML-based Anomaly Detection: The growing adoption of microservices in cloud computing has introduced new challenges for anomaly detection, particularly in real-time cloud environments where services operate with strict latency constraints and dynamic workloads. A key challenge is analyzing distributed traces, which capture execution paths across microservices. These traces are essential for performance monitoring but are difficult to interpret due to frequent updates, evolving dependencies, and high data volumes. Also, the complexity of inter-service dependencies and partial labeling makes real-time anomaly detection even more challenging.

Contextual Logs in ML-based Anomaly Detection: Log anomaly detection is critical for maintaining system reliability, identifying security threats, and enabling failure prediction. This is especially important in real-time cloud environments, where logs evolve dynamically due to frequent system updates, configuration changes, and fluctuating workloads. Services must operate under strict latency constraints while processing massive volumes of log data efficiently. A key challenge is log instability, where constantly changing log structures hinder models from generalizing to unseen patterns. Additionally, the severe class imbalance—where normal log entries vastly outnumber anomalies—causes models to overlook rare but critical events. Moreover, most existing approaches depend on labeled datasets, which are costly and time-consuming to generate, limiting their scalability in real-world deployments.

1.3 Thesis Contributions

This section presents three key contributions in ML-based anomaly detection for real-time cloud environments. Each contribution addresses a unique challenge related to sequential metrics, distributed traces, and contextual logs.

1.3.1 Sequential Metrics in ML-based Anomaly Detection [1][2]

The first contribution addresses ML-based anomaly detection in sequential metric data within real-time cloud environments. We propose a drift adaptation method to address the challenge of concept drift by regularly updating the anomaly detection model to adapt to heterogeneous real-time clouds while meeting time and accuracy requirements. The proposed algorithm incorporates sliding and adaptive window-based methods with a performance-driven approach for effective concept drift adaptation. Building on the Prediction-Driven Anomaly Detection (PDAD-SID) method, we introduce a drift-adapted, real-time anomaly detection algorithm that leverages multi-source prediction for estimating anomaly probabilities and optimizes the model using a Genetic Algorithm (GA). Additionally, our method employs a dynamic algorithm to detect and adapt to concept drift, ensuring the model remains accurate over time in real-time cloud environments. Trace-driven evaluations on three real-world datasets demonstrate that our approach achieves

up to 89.71% accuracy (AUC), outperforming state-of-the-art methods while maintaining efficiency and scalability.

1.3.2 Distributed Traces in ML-based Anomaly Detection [3][4]

The second contribution addresses ML-based anomaly detection in distributed trace data within real-time cloud, where service interactions are highly dynamic. We propose a Trace-Driven Anomaly Detection (TDAD) method that utilizes a Graph Neural Network (GNN) to learn vector representations of traces and employs Positive and Unlabeled (PU) learning to train an anomaly detection model with partially labeled data. By leveraging a small set of labeled anomalous traces alongside a large set of unlabeled traces, this approach facilitates timely and accurate anomaly detection. Building on this, we introduce an Asynchronous Real-Time Federated Learning (ART-FL) framework to mitigate communication and computation overhead in microservice systems. ART-FL enables real-time learning by allowing the global model to proceed asynchronously, without waiting for lagging clients. This ensures continuous computation, with clients performing real-time updates on local streaming data, resulting in improved model convergence compared to synchronized federated learning methods. We conduct a comprehensive experimental evaluation on a real-world microservice benchmark to assess the performance of our proposed method in terms of detection effectiveness, detection time, and resource overhead.

1.3.3 Contextual Logs in ML-based Anomaly Detection [5][6]

The third contribution addresses ML-based anomaly detection in contextual log data within real-time cloud, We propose an Adaptable Log-based Self-supervised method to Catch ANomalies (ALogSCAN). Our approach employs a dual-network architecture consisting of an Auto-Encoder (AE) teacher model and an Encoder-Only (EO) student model leveraging Knowledge Distillation (KD) to provide prompt anomaly detection. In addition, we introduce dynamic filtering and frequency-based reconstruction into our dual-network architecture to adapt to unstable log data and prioritize less common patterns during training. Finally, ALogSCAN relies on self-supervised learning techniques to skip labeling dependency, using the input data as supervision to reach comparable and better accuracy results than other learning methods. We comprehensively evaluate ALogSCAN against six state-of-the-art baselines using real systems' public and private log data. The comparative evaluation demonstrates that ALogSCAN is more efficient in accuracy and detection time than the other anomaly detection approaches. Furthermore, extensive ablation experiments confirm the effectiveness of each principal component in our proposal.

1.4 Background Information

This section provides an overview of key concepts and technologies relevant to the scope of this work. It begins with a discussion of real-time cloud and its role in enabling low-latency applications. Subsequently, the focus shifts to real-time cloud-native, emphasizing modularity, scalability, and resilience achieved through microservices, containerization, and orchestration. The section then explores the fundamentals of ML algorithms and provides an overview of anomalies and anomaly detection techniques.

1.4.1 Real-time Cloud

Real-time cloud refers to cloud-based systems designed to process data and respond to inputs or events within a strictly defined time frame, ensuring predictable and timely outcomes. This capability is crucial for applications where delays can lead to significant issues, such as in smart manufacturing, healthcare, or financial services. According to Ericsson, a leader in telecommunications technology, the real-time cloud integrates components like real-time hypervisors, virtual networks, routers, application execution environments, and development toolchains. These elements work together to manage the edge and cloud continuum, providing end-to-end latency guarantees essential for critical systems [12].

A prominent example of real-time cloud is its application in the Industrial Internet of Things (IIoT), particularly in smart manufacturing. In smart manufacturing environments, IIoT devices such as sensors, actuators, and edge devices continuously generate vast amounts of data. Real-time cloud enables the aggregation, processing, and analysis of this data with minimal latency.

1.4.1.1 Real-time Cloud-native

As clouds has evolved, the focus has shifted toward real-time cloud-native systems, which offer modularity, scalability, and resilience by leveraging a microservices-based architecture. These systems leverage a microservices-based architecture to ensure low-latency processing, seamless scalability, and fault tolerance, enabling real-time decision-making and responsiveness. According to the Cloud Native Computing Foundation (CNCF), cloud-native encompasses the collection of technologies that break down applications into microservices and package them into lightweight containers to be deployed and orchestrated across a variety of servers [13]. In the context of real-time cloud-native systems, these technologies must also support high-velocity data processing, real-time analytics, and rapid adaptation to changing workloads to maintain the performance and reliability of time-critical applications. The key characteristics of real-time cloud-natives are defined as follows:

(A) Microservices: Self-contained, loosely coupled services that execute specific functions, allowing independent development, deployment, and scaling. Microservices use lightweight protocols like RESTful APIs and support agile practices such as CI/CD [13].

A critical concept for managing microservice architectures effectively is observability, which refers to the ability to assess the internal state of a system through its external outputs [14]. Observability ensures the transparency and manageability of distributed systems, making it essential for the reliability and performance of cloud-native applications. The three pillars of observability are as follows:

- Metrics: Quantitative data reflecting the performance and health of services, such as response times and resource utilization. Metrics help ensure microservices operate efficiently and meet SLAs.
- Traces: Visual representations of request journeys across services, offering insights into inter-service dependencies and optimizing overall responsiveness and reliability.
- Logs: Detailed records of events or errors produced by each service. Logs are essential for debugging and diagnosing emergent behaviors, enabling rapid issue resolution.

(B) Containerization: A process that packages applications and their dependencies into portable, isolated units called containers. Tools like Docker ensure consistency across environments, enabling reproducibility and efficient resource utilization [13].

(C) Orchestration: The automated management of containerized microservices, including deployment, scaling, networking, and load balancing. Platforms like Kubernetes streamline operations, ensuring scalability and reliability [13].

1.4.2 Machine Learning

ML is a subset of Artificial Intelligence (AI), which allows machines to learn automatically and improve based on experience without being explicitly programmed [15]. There has been a significant increase in the use of ML algorithms in anomaly detection applications for analyzing data and making decisions [16]. ML algorithms include Support Vector Machine (SVM), K-means, DBSCAN, Decision Tree, Principal Component Analysis (PCA), and Artificial Neural Network (ANN). Additionally, Deep Learning (DL) is a class of ML algorithms based on ANN that uses more than one hidden layer to extract higher-level features defined in terms of lower-level features [17]. DL models are able to catch dependencies between the features of data and can handle high-dimensional data. DL algorithms include Convolution Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short Term Memory (LSTM), Bidirectional Long Short Term Memory (Bi-LSTM) and Autoencoders (AEs). ML techniques can be categorized based on the availability of data labels. There are three main types of ML techniques as follows [8].

- Supervised Learning: This approach uses labeled data to train the ML models. The algorithm observes some example input-output pairs and learns a function that maps from input to output [8].

- Unsupervised Learning: This approach does not require labeled data, making it more versatile. It analyzes data based solely on its inherent structure, typically using distance metrics. The algorithm learns patterns in the input, even though no explicit feedback and guidance are supplied [8]

- Semi-supervised Learning: This approach aims to bring the benefits of both the approaches of supervised and unsupervised techniques. Semi-supervised methods have the underlying assumption that labeled data are available for a portion of data, while the rest of the data are unlabelled [18]. Although this technique, unlike the supervised technique, does not require labeling of all data instances, the need for labeling data for a portion of learning data still makes it less applicable for many environments where such labeling data are not available [8].

- Self-supervised Learning: This approach skips the need for labeled data, using the same input as supervision. This learning method enables transferring representations to other tasks, favoring tasks that depend on the output of a previous process (i.e., downstream tasks), such as natural language processing, computer vision, and anomaly detection [19]. Self-supervised learning broadly divides into two categories: predictive and contrastive. Predictive methods train deep learning models to predict labels derived from the input data, such as masked inputs, statistical properties, or domain knowledge-based targets. Contrastive methods focus on pair-wise discrimination; they apply transformations or augmentations to generate multiple views of data samples and train models to distinguish between view pairs, jointly and independently sampled.

1.4.3 Anomalies and Anomaly Detection

Anomalies are patterns in data that do not conform to a well-defined notion of normal behavior [8]. Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior. Nonconforming patterns in different application domains are called anomalies, outliers, exceptions, aberrations, surprises, peculiarities, or contaminants. Outliers and anomalies are most commonly used in anomaly detection; sometimes, these terms are used interchangeably [8]. Anomaly detection has applications in a broad range of domains, such as fraud detection, network intrusion detection, system monitoring, medical problems, and image and text processing. For instance, in the medical domain, anomalous data observation from an MRI image may show the presence of malignant tumors. Anomalous activity in credit card transactions could be an indicator of credit

card fraud. Also, network traffic with an anomalous pattern may indicate an intrusion into the network [20]. Real-time detection of anomalies is a key requirement of a number of time-sensitive applications, including robotic system monitoring, smart sensor networks, and data center security. Real-time anomaly detection refers to the detection of anomalies within a finite and specified time period. [21].

In contrast, faults are the incapability of a system to perform its necessary tasks that are caused by some abnormal state or bug present in one or multiple parts of a system [20]. Typically, to fix a fault, corrective action is required, such as replacement or maintenance. Based on this notion, we can define fault detection as the problem of finding changes in a system so that it can no longer operate in accordance with the requirements of its user.

1.4.3.1 Types of Anomalies

Anomalies can be classified into the following three categories: [8].

- Point Anomalies: If an individual data instance can be considered as anomalous concerning the rest of the data, then the instance is termed a point anomaly. Let us consider the detection of credit card fraud as an example from real life. Suppose the data set corresponds to an individual's credit card transactions. To simplify the analysis, let us assume that only one feature is used to define the data: the amount spent. A point anomaly is a transaction in which the amount spent exceeds that individual's normal range of expenditures.
- Contextual Anomalies: Anomalies, which can only be identified with respect to a specific context, are called contextual anomalies. This type is most likely to be the most often occurring type when processing real-world data.
- Collective Anomalies: Collective anomalies are the most complex type of anomalies. In this case, multiple instances form an anomaly, whereas each individual instance is not necessarily an anomalous entity. This means that instances in the dataset are related to one another in some way. For example, the number of users' requests increases on a specific day, but it is normal on other days.

In cloud systems, these types of anomalies manifest across different operational and architectural layers. Table 1.1 provides a detailed taxonomy of cloud-specific anomalies. It categorizes anomalies based on their source, behavior, and impact while also linking them to their respective types. This mapping demonstrates how the general anomaly types translate into practical scenarios in cloud environments, offering a structured framework for understanding and detecting such anomalies.

Table 1.1: Taxonomy of cloud anomalies.

Category	**Sub-Category**	**Examples**	**Type of Anomaly**
Source-Based	Application-Level Anomalies	- Software crashes (e.g., memory leaks). - Misconfigurations (e.g., service parameters). - Faulty deployments (e.g., version mismatches).	Point
	Infrastructure-Level Anomalies	- Hardware failures (e.g., disk crashes). - Network issues (e.g., packet loss). - Resource contention (e.g., CPU bottlenecks).	Point
	Security-Related Anomalies	- Unauthorized access. - DoS attacks. - Malware injection.	Collective
Behavior-Based	Performance Anomalies	- Slower response times. - Throughput degradation. - Resource overutilization or underutilization.	Contextual
	Operational Anomalies	- Unexpected workload spikes. - Resource allocation mismatches. - Configuration drifts.	Contextual
	Data Anomalies	- Corrupted logs. - Missing data packets. - Unusual storage patterns.	Point
Impact-Based	Transient Anomalies	- Temporary deviations (e.g., brief CPU spikes). - Often self-correcting.	Contextual
	Persistent Anomalies	- Long-lasting issues (e.g., software bugs). - QoS degradation.	Point
	Cascading Anomalies	- Failures propagating to other services (e.g., database outages).	Collective
	Catastrophic Anomalies	- System-wide failures (e.g., data center crashes).	Collective

1.5 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 outlines the requirements for ML-based anomaly detection in the real-time cloud and critically reviews the state-of-the-art approaches. Chapter 3 introduces an algorithm for concept drift detection and adaptation for anomaly detection in real-time cloud environments. Chapter 4 describes a distributed trace-based anomaly detection algorithm in real-time cloud environments. Chapter 5 presents an algorithm for contextual log anomaly detection in the real-time cloud. Finally, Chapter 6 concludes the thesis and suggests directions for future research in this domain.

Chapter 2

Requirements and Related Work

In this chapter, we first describe an illustrative use case. Next, we use the use case to derive a set of requirements for ML-based anomaly detection in real-time clouds. Finally, we review the related work in light of these requirements.

2.1 Illustrative Use Case

In this section, we present an illustrative use case to derive the general requirements of ML-based anomaly detection in real-time cloud environments.

To better understand the challenges of ML-based anomaly detection in real-time cloud environments, consider a cloud-based real-time video analytics system used for smart traffic management in a metropolitan city. This system relies on a combination of cloud infrastructure and edge computing to analyze high-speed video streams from thousands of traffic cameras deployed across urban intersections. The system detects anomalies such as sudden traffic congestion, accidents, reckless driving, and road hazards. At the core of this system is an ML-based anomaly detection model running within a hybrid cloud-edge environment. The edge nodes—situated near intersections—process video frames in real-time to detect immediate anomalies (e.g., a vehicle running a red light or a pedestrian jaywalking). These localized insights are then transmitted to the central cloud, where they are aggregated, correlated with other data sources (e.g., GPS feeds, weather conditions, historical traffic patterns), and analyzed for broader trends. The cloud layer also plays a critical role in orchestrating large-scale traffic control measures, such as adjusting traffic light sequences, sending alerts to city authorities, or rerouting vehicles dynamically.

However, ensuring accurate and timely anomaly detection in this real-time cloud environment introduces several challenges. Since traffic anomalies must be identified within milliseconds to trigger immediate interventions, such as adjusting traffic light timings or dispatching emergency services, traditional batch-processing models fail to meet the system's stringent low-latency constraints. Additionally, the continuous influx of high-velocity video streams means that data distributions frequently shift due to varying weather

conditions, seasonal traffic fluctuations, and infrastructural changes. Without adaptation mechanisms, anomaly detection models trained on historical data risk becoming outdated, leading to misclassifications, false alarms, or missed incidents.

Compounding these challenges, the system must dynamically adjust to fluctuations in workload as the number of active cameras changes based on traffic density and system demand. Anomaly detection models must therefore scale seamlessly to maintain efficiency without compromising detection performance. Moreover, the diverse nature of video data—affected by variations in lighting, camera angles, and environmental conditions—demands robust ML models capable of generalizing across highly variable inputs.

2.2 Requirements

In this section, we outline the general requirements for ML-based anomaly detection in real-time cloud environments, considering the illustrative use case. We then present solution-specific requirements addressing the challenges identified in Section 1.2.

2.2.1 General Requirements for ML-based Anomaly Detection in Clouds

The following requirements are identified as general requirements for the problem of anomaly detection in a real-time cloud-based smart manufacturing environment.

Accuracy: The anomaly detection model must achieve high precision and recall, minimizing false positives and false negatives. For instance, in the smart traffic management system, the model must correctly distinguish between genuine accidents and routine traffic slowdowns. High false positives could trigger unnecessary emergency dispatches, while false negatives could delay responses to critical incidents.

Real-Time Detection: The model must process high-velocity streaming data with minimal latency. In the smart traffic system, detecting a traffic accident even a few seconds late could delay response times and worsen congestion. The system must ensure sub-second anomaly detection to enable timely interventions.

Scalability: The system must efficiently handle thousands of concurrent data streams from cameras across the city. The anomaly detection pipeline must scale horizontally as new cameras are added, ensuring consistent performance without overwhelming cloud resources.

Adaptability: The model must adapt to evolving data distributions caused by environmental changes (e.g., day vs. night traffic patterns, seasonal shifts, construction projects). The ability to detect concept drift and update models dynamically ensures that the system remains accurate under changing conditions.

2.2.2 Requirements for Sequential Metrics in ML-based Anomaly Detection

While the requirements identified in Section 2.2.1 were general to ML anomaly detection models, in this section, we present the requirements specific to a solution that adapts to concept drift in sequential metrics data while detecting anomalies in real-time clouds.

Autonomy: The adaptation algorithm must operate autonomously with minimal or no human intervention. Real-time cloud systems involve diverse anomaly detection models with varying architectures, which can experience degraded performance due to concept drift. An automated solution is essential for continuously monitoring and adapting these models to maintain effective anomaly detection without manual oversight.

Concept Drift Adaptation: The algorithm must be capable of accurately detecting concept drifts with varying degrees of severity and frequency. In real-time cloud systems, concept drifts can manifest gradually or abruptly, impacting anomaly detection differently. An adaptation mechanism should differentiate between these drift types and respond appropriately, ensuring both timely and effective adaptation without overreacting to minor or temporary fluctuations.

Post-Adaptation Detection Accuracy: The algorithm should ensure that the anomaly detection model maintains high accuracy after adaptation. The performance post-adaptation must remain comparable to or exceed the pre-drift performance, enabling reliable detection of anomalies even under evolving cloud conditions.

Adaptation Time Efficiency: Adaptation to concept drift must be performed swiftly to minimize the impact on real-time anomaly detection. A short adaptation time ensures that the detection system can quickly recover from drift and continue identifying anomalies effectively.

2.2.3 Requirements for Distributed Traces in ML-based Anomaly Detection

In this section, we present the key requirements for effectively handling distributed trace data in ML-based anomaly detection.

Integration of Multi-Source Data: The system should seamlessly integrate data from various sources, such as logs and traces, to detect anomalies. This integration is critical as it allows for a comprehensive view of interactions and the internal state of services, capturing details like memory usage, request handling, and response times.

Preservation of Trace Structure: Anomaly detection methods must maintain the hierarchical, parallel, and asynchronous structure of traces. Simplifying these into sequential representations can obscure critical causal relationships and temporal dynamics that are essential for accurately detecting anomalies.

Scalability to Trace Size: The system should effectively handle traces of varying

lengths and complexities, including very large traces generated by complex workflows. It must process and analyze these traces without a significant increase in computational overhead or degradation in detection performance.

Low Communication Overhead: The anomaly detection system must leverage collaborative learning to analyze data across distributed nodes while minimizing communication rounds between clients and the central server to reduce bandwidth usage and latency.

2.2.4 Requirements for Contextual Logs in ML-based Anomaly Detection

In this section, we present the key requirements for effective ML-based anomaly detection systems utilizing contextual log data.

Contextual Detection: ML-based anomaly detection should analyze logs in context rather than in isolation. The meaning and significance of a log entry depend on its relationship with other logs, system state, and service interactions. Considering semantic similarities, temporal patterns, and cross-service dependencies helps improve detection accuracy and reduces false positives . For example, in OpenStack Cinder logs, "Volume attachment failed due to connection timeout" and "Failed to attach storage due to network timeout" are semantically similar but written differently.

Adaptability to Log Format Variations: Cloud environments are highly dynamic, with frequent modifications, additions, or removals of logging statements across different software versions. As log formats evolve over time, static anomaly detection models become less effective. To maintain high detection accuracy, the anomaly detection system must adapt dynamically to changes in log patterns without manual intervention.

Effective on unlabeled data: Manual labeling of logs is often impractical and time-consuming, especially in large-scale real-time cloud environments. Since most log data is unlabeled, ML-based anomaly detection must effectively identify anomalies without relying on labeled training data.

Handling High Log Volume: Since large-scale real-time cloud environments generate a vast amount of log data, the anomaly detection solution should be scalable. For example, As OpenStack environments generate millions of logs per hour, anomaly detection models must Scale to handle distributed log streams across cloud nodes.

2.3 Related Work

In this section, we present the state-of-the-art for the challenges identified in this thesis. First, we review the existing ML-based anomaly detection solutions in clouds. Next, we review the prior art on the three main contributions of this thesis, i.e., sequential metrics in ML-based anomaly detection, distributed traces in ML-based anomaly detection, and contextual logs in ML-based anomaly detection.

2.3.1 ML-based Anomaly Detection in Clouds [7]

In this section, we review the research work on ML-based anomaly detection in cloud environments. Existing anomaly detection methods can be classified into three categories (A) statistical, (B) machine learning, and (C) deep learning methods.

(A) Statistical Methods: Statistical anomaly detection primarily uses distance-based methods, which calculate anomaly scores through k-nearest-neighbors distance [22], local outlier factor (LOF) [23], or histogram-based outlier scores [24]. Common models include the autoregressive model, moving average, and autoregressive moving average models [25]. While effective for consistent trends, these models are noise-sensitive and assume known data distribution, which is often not valid for high-dimensional datasets with unpredictable events. Histogram-based methods are inadequate for high-dimensional data due to their inability to analyze feature interactions. Additionally, selecting the optimal k value for accurate LOF computation in the LOF algorithm [23] is challenging.

(B) Machine Learning Methods: Machine learning techniques can be supervised, unsupervised, or semi-supervised [26]. Supervised learning [27, 28] involves training models with labeled data, which typically requires human input. This, however, can be time-consuming and sometimes infeasible in practical scenarios, where some anomalies may be unknown.

Alternatively, unsupervised learning methods for anomaly detection typically aim to identify the unique features which help distinguish abnormal data from normal data. To this end, iForest [29] uses a forest of isolation trees from training samples to compute anomaly scores based on path lengths. iForest is efficient for large and high-dimensional data, though its detection performance reaches convergence rapidly with a limited number of trees. In [30], an unsupervised clustering-based method called Online Anomaly Detection in Data Streams (ODS) was introduced for real-time anomaly detection in telemetry data. However, it may easily be trapped in local minima and do not behave proactively when changes occur.

In semi-supervised learning, the data distribution is estimated from normal behavior during training, and then the test data is compared to the obtained distribution. Hu et al. [31] developed six meta-features for univariate and multivariate time series using a One-Class Support Vector Machine (OCSVM) for anomaly detection. OCSVM, however, is sensitive to outliers in the absence of labels.

(C) Deep Learning Methods: Munir et al. [32] introduced DeepAnT, which relies on deep Convolutional Neural Network (CNN). This method is effective for anomaly detection in time-series data, even with small datasets. In [33], a multilayer convolutional recurrent autoencoder is developed for detecting anomalies in multivariate time series. Zhou

et al.[34] developed a variational LSTM (VLSTM) model for unbalanced, high-dimensional industrial data. The authors of [35] proposed an LSTM-Gauss-NBayes method, which combines LSTM-NN and Naive Bayes for anomaly detection in IIoT. A real-time anomaly detection method for NoSQL systems called RADAR was proposed in [36], which identifies anomalous events by extracting process information during resource monitoring. Malhorta et al. [37] used a multiple-prediction technique assessing anomalies through prediction error and Gaussian error distribution. Wang et al. [38] proposed an improved LSTM method to detect anomalies in time series data by forecasting and comparing predicted and observed sequences, addressing the issue of data with diverse distributions and the lack of labeled anomalous data.

Table 2.1 compares the existing relevant works for sequential metric anomaly detection based on the general requirements. Existing studies simply detect anomalies in streaming data, and they mostly ignore real-time detection; for example, some works, e.g., [29], [32], store the observed data for processing and do not stick to the one-pass criterion. Therefore, they fail to deliver meaningful results in a timely manner. Furthermore, some works (e.g. [1], [30], [32], [29]) fail to incorporate adaptability, crucial for addressing the changing statistical behavior of data streams.

Table 2.1: Comparison of the related work on ML-based Anomaly Detection in Real-Time Cloud.

Related Work	Requirements			
	Accuracy	Real-Time Detection	Adaptability	Scalability
[23, 24, 25]	✓	✗	✗	✗
[29], [31]	✓	✗	✗	✓
[32], [35], [34]	✓	✗	✗	✗
[36], [38]	✗	✓	✗	✗
[30], [33]	✓	✗	✓	✓
[1]	✓	✓	✗	✗

2.3.2 Sequential Metrics in ML-based Anomaly Detection

In the following, we review the related work on anomaly detection, followed by existing methods addressing challenges in sequential metrics within real-time clouds, such as concept drift.

Mothukuri et al. [39] proposed an anomaly detection method using Gated Recurrent Units (GRUs) models for the real-time and proactive identification of network intrusions in IoT systems. Some studies have studied the problem of anomaly detection in the presence of concept drift. Dromard et al. [40] presented ORUNADA, an unsupervised network anomaly detection algorithm using incremental grid clustering and a discrete sliding window to update features. Spinosa et al. [41] introduced OLINDDA, an unsupervised clustering algorithm for detecting anomalies in streaming data by modeling normal data

with a hypersphere around normal data clusters. Yu et al. [42] developed a framework for detecting anomalies in large-scale datasets, clustering nodes based on geographic location and network topology, and employing a two-phase majority voting algorithm to identify anomalous nodes. This method effectively detects new anomalies by spotting deviations from majority behavior, but its adaptability to system behavior changes during runtime is unclear.

In [43], the so-called A-Detection method was proposed, which uses reservoir sampling and singular value decomposition (SVD) to analyze data streams and applies Jensen Shannon (JS) divergence to detect anomalies. Yang et al. [44] introduced ASTREAM, an anomaly detection system for data streams in real-time scenarios. ASTREAM incorporates sliding windows, concept drift detection, and updating models in a hashing-based locality-sensitive iForest model to address the challenges of continuous data streams. Presenting a comprehensive performance evaluation using the KDDCup99 dataset, the authors of [44] have shown that the ASTREAM algorithm is robust in handling the challenges of IIoT data streams, such as managing infinite data and adapting to data distribution changes.

In [45], a deep RNN-based method called Online RNN-AD was introduced for online time-series anomaly detection. This method incorporates local normalization of incoming data and incremental neural network retraining, showing adaptability to concept drift in time-series data. In [46], an online and adaptive anomaly detection model was proposed by employing a one-class SVM, which uses unlabeled data to create a hyperplane, isolating a region with most normal vectors, and evaluates anomalies based on their proximity to this hyperplane. This method allows dynamic input normalization and adaptation based on operator feedback, offering a confidence level for each anomaly. We note that recalculating the hyperplane for conflicting operator feedback can hamper real-time accuracy. Besides, the SVM approach suffers from the curse of dimensionality and struggles with large feature sets.

Table 2.2 compares the existing relevant works for sequential metric anomaly detection based on the specific requirements. The general requirements of ML-based anomaly detection models are not included in the table.

Table 2.2: Comparison of related work concerning the requirements for sequential metric anomaly detection.

Related Work	Requirements			
	Autonomy	Concept Drift Adaptation	Post-Adaptation Detection Accuracy	Adaptation Time Efficiency
ORUNADA [40]	✗	✓	✗	✗
OLINDDA [41]	✗	✓	✓	✗
[42]	✓	✓	✗	✗
A-Detection [43]	✗	✓	✓	✗
ASTREAM [44]	✓	✓	✗	✓
Online RNN-AD [45]	✗	✓	✓	✓

2.3.3 Distributed Traces in ML-based Anomaly Detection

In this section, we review existing approaches for anomaly detection in centralized learning and federated learning across various applications and domains.

2.3.3.1 Centralized Learning Anomaly Detection

Gan et al. [47] developed Seer, a performance anomaly detection method for services that integrates convolutional and LSTM layers. In [48], dual neural networks for service anomaly detection were proposed. Their method involves a variational autoencoder, trained on normal activity to spot anomalies through reconstruction errors, and a convolutional neural network trained on failure-injected data for specific anomaly detection. This approach effectively reduces false positives and identifies anomaly types in anomalous services. Bogatinovski et al. [49] developed a self-supervised encoder-decoder network for anomaly detection by learning event dependencies. In [14], SCWarn employed a multi-model LSTM approach to analyze data from diverse multi-source inputs.

The authors of [50] proposed TraceAnomaly, which utilizes a deep Bayesian neural network with posterior flow for anomaly detection. Nedelkoski et al. [51] proposed a multi-modal LSTM neural network for anomaly detection in multi-service applications trained on normal execution traces. Microscope [52] is a performance anomaly detection model for applications, tracking front-end KPIs of microservices against predefined Service Level Objectives (SLOs). Jin et al. [53] proposed their so-called RPCA, which is an offline anomaly detection method for microservices using distributed traces. Centralized methods for anomaly detection, while effective, face practical challenges in distributed edge device scenarios. First, these models often lose accuracy over time due to their inability to adapt to evolving data. Second, privacy concerns hinder data exchange among edge devices, leading to the creation of the so-called “data islands," which impairs anomaly detection efficiency. Third, centralized machine learning methods are not able to satisfy both the accuracy and real-time requirements of anomaly detection.

2.3.3.2 Federated Learning Anomaly Detection

Nguyen et al. [54] used FL for network intrusion detection in IoT devices. In [55], the authors used FL and LSTM to monitor activities and detect abnormal energy usage in smart homes. Truong et al. [56] developed a fast, resource-efficient FL method for anomaly detection in industrial control systems. Mothukuri et al. [39] employed FL with federated GRU models for detecting and classifying attacks in IoT networks. In [57], the authors introduced an FL-based system for anomaly detection in IoT streaming data, incorporating a compression algorithm to improve FL gradient communication efficiency and reduce network resource use. In [58], an FL-based anomaly detection method using autoencoder architecture was proposed for IIoT. In [59], an FL method for IoT anomaly detection was proposed, which

integrates differential privacy for enhanced user privacy. Pei et al. [60] presented an anomaly detection framework that utilizes LSTM for identifying anomalies in network traffic. In [61], an FL-based approach named FLOG was proposed to detect anomalies in distributed log data. Deepfed, a FL-based framework for intrusion detection in cyber-physical systems (CPSs), was introduced in [62]. CNNs and GRUs are combined in this architecture to detect threats. Li et al. [63] applied deep learning for extracting features from log sequences and employed an FL framework for training in a distributed IoT environment.

FedAnomaly, an FL-based transformer framework, was proposed in [64] for anomaly detection in cloud manufacturing. This method enables distributed edge devices to jointly train a global model without sharing data, thus preserving privacy and enhancing anomaly detection in cloud manufacturing settings. Kong et al. [65] introduced a federated graph anomaly detection framework that leverages contrastive learning and anomaly information updates to enhance detection performance on distributed graph data. Jithish et al. [66] developed an FL approach for anomaly detection in smart grids, focusing on enhancing data security and privacy.

Table 2.3 compares the existing relevant works for distributed traces anomaly detection based on our requirements. The general requirements of ML-based anomaly detection models are not included in the table. Centralized learning in microservices uses supervised or unsupervised algorithms [47, 48, 49, 50, 52]. Unsupervised learning, assuming most data is normal, fails to effectively incorporate historical anomalies, while supervised learning requires extensive, resource-heavy labeling, potentially missing diverse anomalies. Additionally, trace comparison methods are useful for anomaly detection but may not meet real-time detection needs due to their slowness [67], [68]. While existing research uses deep learning for metric or log analysis in microservice anomaly detection, it often neglects distributed trace data's potential [50, 51, 52]. These studies simplify traces to sequences of service calls, ignoring their complex structure, including hierarchical, parallel, and asynchronous relationships, and the detailed log messages critical for anomaly detection. Furthermore, current log anomaly detection methods [69, 70] mainly learn from normal operations, identifying deviations as anomalies without capturing the subtle anomaly patterns effectively. The existing FL approaches [64, 66] assume uniform device behavior under synchronous protocols, which does not accurately reflect the real-world heterogeneity of devices. These approaches struggle with variations in data volume and distribution, system latency, hardware configurations (such as memory and processor speed), and the availability issue of edge devices. This leads to inconsistent computation times and contributions across clients in the FL model.

2.3.4 Contextual Logs in ML-based Anomaly Detection

This section reviews existing anomaly detection methods, grouping them by the learning approach: supervised, unsupervised, and semi-supervised.

Table 2.3: Comparison of related work concerning the requirements for distributed traces anomaly detection.

Related Work	Domain	Data Type	Learning Manner	Requirements			
				Multi-source data	Preservation of Trace Structure	Scalable to Trace size	Low communication Overhead
DIoT[54]	Intent of Things (IoT)	Metric	FL	✗	✓	✓	✓
FSLSTM[55]	Smart Building	Log	FL	✗	✓	✗	✓
FL-GRU[39]	Industrial Control System	Metric	FL	✗	✓	✓	✓
FATRAF[56]	Industrial Control System	Metric	FL	✗	✗	✓	✓
FLOGCNN[61]	Software Systems	Log	FL	✗	✓	✗	✓
DeepFed[62]	Cyber Physical System	Metric	FL	✗	✓	✗	✓
FedAnomaly [64]	Cloud Manufacturing	Metric	FL	✓	✗	✗	✓
TCN-ACNN[63]	IoT	Log	FL	✓	✗	✗	✓
TraceAnomaly [50]	Microservice Application	Trace	CL	✗	✗	✓	✗
MultimodalTrace [51]	Microservice Application	Trace	CL	✗	✗	✓	✗
LogAnomaly [70]	Software Systems	Log	FL	✗	✓	✗	✓
RobustLog [69]	Software Systems	Log	CL	✗	✓	✓	✗
Microscope [52]	Microservice Application	Trace	CL	✓	✓	✓	✗
FL-SmartGrid-AD[66]	Smart Grid	Metric	FL	✓	✓	✗	✓
TDAD [3]	Microservice Application	Trace	CL	✓	✓	✗	✗
SCWarn [14]	Microservice Application	Metric and Log	FL	✓	✓	✗	✓
Seer[47]	Microservice Application	Metric and Trace	CL	✗	✓	✓	✗

2.3.4.1 Supervised anomaly detection

Supervised anomaly detection has been extensively studied by many research works that apply different supervised learning techniques on labeled data. However, most of them fail to manage log instability. LogRobust [69] was the earliest supervised method to address this issue. First, LogRobust generates vector representations of log templates by leveraging word embedding learning (i.e., FastText) and Term Frequency-Inverse Document Frequency (TF-IDF). Then, it implements an attention-based bidirectional LSTM model to capture contextual information within log sequences and automatically learn the importance of different log events. These components enable LogRobust to detect anomalies even in unstable log data. Similarly, HitAnomaly [71] tackles log instability using a hierarchical transformer structure that effectively models log template sequences and parameter values.

Nevertheless, supervised methods rely on labeled datasets, requiring manual intervention to label normal and abnormal events. This manual effort creates extensive overhead in terms of time, resources, and data management, impeding the timely operation of supervised methods. In addition, LogRobust and HitAnomaly lack class imbalance support, being prone to reduced accuracy when the number of anomalies is much less than normal data.

2.3.4.2 Unsupervised anomaly detection

Unsupervised anomaly detection skips the need for labeled data and provides solutions that effectively handle class imbalance. The anomaly detection model in [72] combines AE networks and Isolation Forest (AE+IF) to handle feature extraction and detect anomalies based on the extracted features, respectively. In contrast, DeepLog [73] and LogAnomaly [70] work with log sequences that use indexed event templates. Both approaches employ LSTM networks to predict the next log event index, flagging deviations as anomalies. LogAnomaly goes a step further, incorporating natural language processing to

Table 2.4: Comparison of related work concerning the requirements for contextual log anomaly detection.

Related Work	Learning approach	Requirements			
		Contextual Detection	Adaptability to Log Format	Effective on Unlabeled Data	Handling High Log Volume
HitAnomaly [71]	Supervised	✗	✓	✗	✓
LogRobust [69]		✗	✓	✗	✓
AE+IF [72]	Unsupervised	✓	✗	✓	✓
LogAnomaly [70]		✗	✗	✓	✓
DeepLog [73]		✗	✗	✓	✓
PLELog [74]	Semi-supervised	✗	✗	✓	✓
LogBERT [75]		✗	✗	✓	✓

generate log template sequence vectors and log count vectors. These vector representations capture semantic and syntactic information of log data and feed the training of the LSTM network.

However, unsupervised methods tend to produce low accuracy performance due to the lack of supervision, particularly when dealing with unstable log data. In addition, methods based on LSTM networks struggle to meet strict time requirements due to their computationally expensive nature, especially for long sequences. LSTM process log events sequentially, one at a time, which leads to delays when managing large volumes of logs.

2.3.4.3 Semi-supervised anomaly detection

Semi-supervised anomaly detection includes different solutions requiring minimal labeled data. PLELog [74] employs a probabilistic label estimation approach, applying positive and unlabeled learning on known normal sequences in the training set to derive pseudo-labels for unlabeled log sequences. These pseudo-labels feed the training of a supervised deep-learning model that classifies log sequences as normal and abnormal. However, PLELog's performance heavily depends on the accuracy of the probabilistic label estimation, which is influenced by the data characteristics. Therefore, unstable log data can lead to poor accuracy performance.

LogBERT [75] leverages Bidirectional Encoder Representations from Transformers (BERT) to capture the contextual embeddings of log entries within the broader sequence. It employs two self-supervised training tasks, masked log key prediction and volume of hyper-sphere minimization, to learn patterns of normal log sequences. However, LogBERT disregards the semantic information of anomalous logs. In addition, this semi-supervised method performs multiple predictions on each sequence, which is unsuitable to meet strict time requirements.

Table 2.4 summarizes the comparison the reviewed log anomaly detection methods based on the key requirements discussed in Section 2.2.4. In general, existing anomaly detection methods fail to meet all the four requirements. Supervised methods like HitAnomaly and LogRobust address log instability and provide scalability support. However, they rely

on extensive batch processing and fail to work with unlabeled data, limiting their timely execution in dynamic cloud environments.

In contrast, unsupervised (i.e., AE+IF, LogAnomaly, DeepLog) and semi-supervised (i.e., PLELog, LogBERT) methods operate with non- or minimal-labeled data and leverage techniques that support scalability, such as AE+IF's isolation forests and LogBERT's transformer-based architecture. However, these methods fail to effectively adapt to evolving log patterns. For example, LogBERT requires retraining to model unseen log data. Furthermore, unsupervised methods based on LSTMs (i.e., LogAnomaly, DeepLog) are prone to delays due to their costly sequential processing of log events. Semi-supervised methods also struggle to meet strict time conditions due to their extensive training and prediction times. The research gap described in this section highlights the need for prompt, adaptive, and scalable anomaly detection in dynamic cloud environments, particularly when dealing with large volumes of unlabeled log data.

2.4 Conclusion

This chapter outlined an illustrative use case and derived key requirements for ML-based anomaly detection in real-time cloud environments. We then reviewed existing algorithms and evaluated them against these requirements. Our analysis revealed that current anomaly detection methods fail to fully address the three key challenges discussed in Section 1.2.

Chapter 3

Sequential Metrics in ML-based Anomaly Detection[1]

3.1 Introduction

ML-based anomaly detection in real-time cloud environments relies heavily on sequential performance metrics such as latency, throughput, and resource utilization. These metrics, which evolve dynamically over time, pose significant challenges for anomaly detection models due to their inherent temporal dependencies, high data velocity, and the need for low-latency decision-making. Among these challenges, one of the most critical is concept drift, where shifts in the statistical properties of data streams can degrade model accuracy over time [11]. Unlike static datasets, sequential metrics in real-time cloud environments demand continuous adaptation to ensure reliable anomaly detection. Real-time cloud systems introduce additional complexities due to their large-scale, high-frequency data streams and stringent latency constraints. Adapting ML models in such settings is particularly challenging, as it requires real-time processing of continuous sequential data while maintaining accuracy and efficiency. Moreover, time series characteristics such as periodicity, seasonality, and feature correlations are intensified in real-time cloud environments, further complicating anomaly detection. Many existing ML-based approaches, originally designed for general-purpose cloud environments, struggle to address these challenges, as they often lack mechanisms for handling high-frequency sequential data or adapting to concept drift in real time.

This chapter explores these challenges, focusing on the role of sequential metrics in ML-based anomaly detection for real-time cloud environments. We analyze how sequential data

[1]This chapter is based on two published papers: [1] Mahsa Raeiszadeh, A. Saleem, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "A deep learning approach for real-time application-level anomaly detection in IoT data streaming," in Proc. *IEEE Consumer Communications & Networking Conference (CCNC)*, pp. 449–454, 2023 and [2] Mahsa Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "Real-Time Adaptive Anomaly Detection in Industrial IoT Environments," in *IEEE Transactions on Network and Service Management (TNSM)*, vol. 21, no. 6, pp. 6839–6856, 2024.

behavior influences anomaly detection performance and discuss the limitations of current methods in adapting to evolving cloud conditions. By addressing these issues, we aim to highlight the need for adaptive, low-latency, and robust ML models that can effectively detect anomalies in real-time clouds.

The rest of this chapter is organized as follows. First, we present an illustrative use case that highlights the specific challenges and importance of addressing concept drift in real-time cloud anomaly detection. Next, we detail the proposed algorithm, followed by its evaluation and comparison with existing methods. Finally, we summarize the findings and contributions of this chapter.

3.2 Illustrative Use case

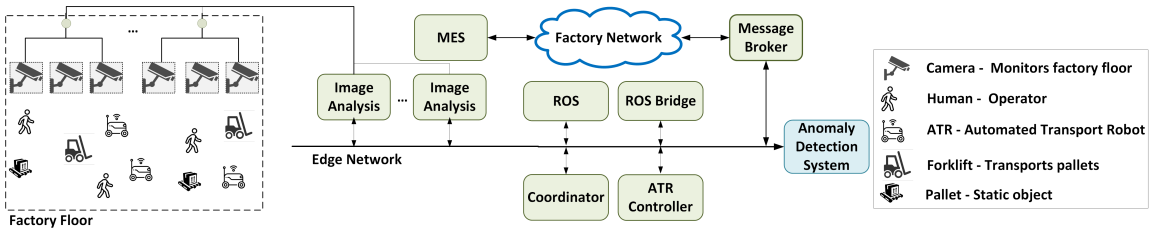


Figure 3.1: Illustration of a cloud-based smart manufacturing environment.

In this section, we present an illustrative use case. Fig. 3.1 depicts a cloud-based smart manufacturing environment where Automated Transport Robots (ATRs) navigate a factory floor using camera-based positioning. This serves as a motivating example to highlight real-time cloud constraints and the need for an ML-based anomaly detection system. As shown in Fig. 3.1, camera-based positioning is used to control the ATRs on the factory floor. Cameras covering the factory floor are connected via a Local Area Network (LAN). To avoid any blind spots, neighboring cameras have overlapping coverage. Several cameras are mounted as visual sensors to capture video from the factory floor. The captured video streams are processed to locate and identify mobile objects and/or obstacles, building a real-time world model that can be used for path planning and obstacle avoidance. The planned paths are sent in segments to the ATRs by the ATR Controller. The Robot Operating System (ROS) is used for robot control. Given that both ROS v.1 and ROS v.2 nodes coexist, ROS Bridge is used to translate from ROS v.1 to ROS v.2. A number of tasks need to be accomplished partly in parallel, like scheduling the fleet of ATRs and the Manufacturing Execution System (MES). One observation of broker-based messaging systems is that they support some of the service meshes' functionality. For example, they can provide basic load balancing using their queue and consumer groups. The smart manufacturing system requires latency to control ATRs.

This cloud-based environment qualifies as a real-time cloud because it processes and

responds to critical data within bounded time constraints. Real-time capabilities are essential for latency-sensitive tasks such as controlling ATRs, processing continuous video streams for path planning and obstacle avoidance, and delivering prioritized anomaly alerts. Moreover, its event-driven architecture and ability to adapt dynamically to changes, such as replacing IoT devices or handling anomalous network events, ensure the system meets the stringent requirements of real-time industrial operations. Anomalous behaviors, such as ATRs deviating from their paths or moving without updated instructions, pose significant risks in this environment.

To further elaborate on this use case, let us provide an example in this setting. In a highly automated manufacturing environment, the ATRs are essential for the seamless transport of materials across the factory floor, and they are scheduled to deliver critical components to assembly lines within stringent time windows. This precision is necessary to maintain continuous production without delays, where any disruption could lead to substantial downtime costs and production delays. During routine operations, it is conceivable that an ATR might deviate from its scheduled path due to a navigation error or encounter an unexpected obstacle. For instance, if an ATR accidentally enters an area restricted for ongoing maintenance, it risks collisions or damage to sensitive equipment, potentially causing safety hazards or costly interruptions. Cameras continuously stream data to the edge network for preliminary analysis, including monitoring the ATRs' movements. An advanced anomaly detection system, running on the real-time cloud, processes this data to identify any deviations from normal operational patterns in a bounded time. Upon detecting an anomaly, such as an ATR moving off its planned path or stopping unexpectedly, the system sends an immediate alert to the ATR controller and the factory's central monitoring system. These alerts are prioritized to ensure they are processed in a timely manner, adhering to the real-time needs of the smart manufacturing environment. The ATR controller may initiate corrective actions based on the specific anomaly detected. This might involve rerouting the ATR automatically or stopping it before entering a restricted area and, if necessary, dispatching a human operator for manual intervention. Additionally, the feedback concerning the anomaly is used to refine and enhance the detection algorithms, thereby improving the system's predictive capabilities and preventing future incidents.

3.3 System Model

Fig. 3.2 illustrates a smart factory environment, which comprises a factory floor, an edge network, and a remote cloud. The factory floor comprises human operators, dynamic objects such as ATRs and forklifts, and static obstacles like pallets. ATRs are assisted by visual object detection in order to pick up the components without human assistance and deliver them to the production line. Multiple cameras, each equipped with an Image Analysis module, are mounted in the area. Each camera transmits the captured video to

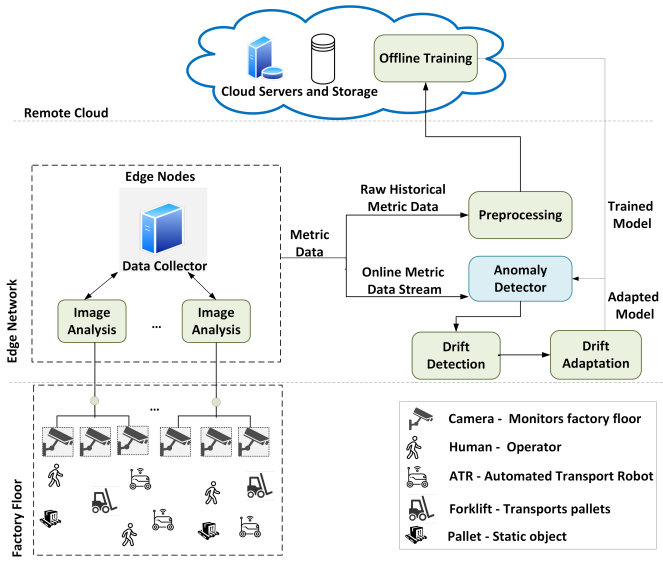


Figure 3.2: System Model

the Image Analysis module via REST API, which is typically used in industrial systems for inter-component communication [76]. The captured video streams are analyzed in the Image Analysis module to detect, classify, identify, position, and track the objects on the factory floor to build a world model that can be used for objects' traveling, scheduling, and obstruction avoidance. For instance, scheduling ATRs' travel paths involves planning how ATRs should travel to arrive at the destination at a certain time and is based on a set of objects with predicted trajectories.

Almost all of the processing, such as travel planning and scheduling, is carried out in the edge network rather than the devices on the factory floor. As shown in Fig. 3.2, the data collector in the edge network has the crucial task of collecting the data in the form of time-series metric data. The time-series metric data can be either one-dimensional, where a sequence of measurements from the same features are collected, or multi-dimensional, where a sequence of measurements from multiple features or sensors are collected. The smart factory system may generate multi-dimensional series metric data with intricate features such as ATRs status, which is stationary or moving, or the wheel velocity of ATRs. The raw collected metric data is passed to the Preprocessing module to clean and normalize. The preprocessed data undergoes offline training to develop a time-series prediction model. Subsequently, the trained model is deployed in the Anomaly Detector system, which is responsible for detecting application-level anomalies. Moreover, the data collector also receives an online metric data stream and feeds it to the Anomaly Detector in order to carry out a real-time detection of anomalies.

Ideally, detecting anomalies should be as simple as computing the distance between the ground truth (i.e., class labels) and the target (i.e., record). A record is a data sample at

a specific moment within a series. A record can take the form of a vector within a multi-dimensional time-series data or a scalar value within a one-dimensional time-series data. However, data labels may not always be available in practice since labeling the data is often challenging, mainly because anomalous records (i.e., records that do not conform to normal or expected records) are rare compared to normal records. To cope with this, a prediction-driven anomaly detection mechanism is needed, where the ground truth is replaced by their predicted values, which can be generated by a time-series prediction algorithm. Clearly, such an approach is dependent on the accuracy of the prediction module, which is usually affected by the anomalous records in the historical data.

Due to the dynamic nature of configurations and workloads caused by, for instance, updates to ROS or replacement of ATRs, the distribution of incoming data may undergo frequent changes over time, resulting in concept drift. This, in turn, leads to false anomaly detection and results in poor performance of the detection model. To address this issue, the Drift Detector within the edge network takes in the anomaly rate (which specifies the likelihood that an anomaly has occurred) as the input. If the anomaly rate consistently increases, the Drift Detector identifies a concept drift and triggers an alert to indicate drift occurrence. Following a drift alert, the system collects new data, and the Drift Adapter in the edge network adjusts the anomaly detection model to adapt to the concept drift using the newly received data.

In the smart factory environment, anomaly detection is crucial for addressing a wide range of cybersecurity and physical vulnerabilities, particularly in industrial control systems. Key concerns include data interception and tampering, where IIoT data streams are vulnerable to manipulation by external attackers, potentially hiding real anomalies or creating false ones. This risk extends to unauthorized access and control over critical IIoT systems, leading to failures such as operational disruptions and data breaches, which often result from root causes such as software vulnerabilities, insecure network configurations, and/or compromised third-party services. Network-based threats such as DDoS attacks disrupt services and pose safety hazards, representing another failure mode. Insider threats involving system users compromising IIoT security and physical security breaches, such as tampering with IIoT devices, are also key components of the threat landscape. Also, failures like application crashes due to root causes such as software bugs or resource leaks can cause significant operational downtime. Similarly, failures such as network latency, commonly caused by root causes like bandwidth overload or hardware malfunctions, can significantly disrupt manufacturing operations. In this paper, we make the following assumptions:

- We assume that the smart manufacturing environment is equipped with automated network management, such as a self-healing system [77].
- We assume that the data collected by different types of industrial sensors on the factory floor is transmitted to edge nodes through REST API, which is typically used in industrial systems for inter-component communication [76].

- We assume that IoT devices generate a high-volume data stream with high frequency that is not independent and identically distributed (non-i.i.d.) and possesses high correlation. Additionally, this data varies in format and characteristics, often including irrelevant, noisy, or redundant elements [78].
- We assume that data patterns change over time, due to a phenomenon commonly known as concept drift, which can manifest in various forms. We specifically consider three types of concept drifts, namely, sudden, gradual, and recurring [79].
- We assume that various types of anomalies may happen in the smart manufacturing environment. These include hardware anomalies (e.g., caused by environmental interference, device malfunction, or reading errors) and software anomalies (e.g., resulting from program exceptions, transmission errors, or malicious attacks) [77].

3.4 Problem Formulation

In cloud environments, a significant portion of the data generated consists of time-series data exhibiting temporal correlations, where a record collected at a one-time point may have connections to previously collected data records. Time-series data consist of successive observations collected in chronological order at each time slot t. In our study, we define the stream of records as $R = \{\mathbf{r}_0, \mathbf{r}_1, ..., \mathbf{r}_M\}$, $(\mathbf{r}_i \in \mathbb{R}, i \in [0, M])$, from M time instants and each data record $\mathbf{r}_i$ is an D-dimension vector.

In the environment described in Section 3.3, considering the diversity of architectures employed in the edge networks, the varying requirements for time-sensitive smart factory services, and the variety of drift types, the task of detecting anomalies in real time and coping with drifts while keeping all possible variations is challenging. The objective of this work is to determine the real-time anomaly status of an observation $\mathbf{r}$ at time slot t, taking into account the temporal changes in the data distribution over time.

3.5 Proposed Solution

To tackle the problem described in Section 3.4, we propose an approach to effectively cope with the concept drift of data streams and accomplish real-time and accurate anomaly detection. Fig. 3.3 shows the overview of our proposed Scalable and Adaptable Prediction-Driven Anomaly Detection (SAPDAD) method, which comprises two main phases: (i) offline phase and (ii) real-time phase. During the offline phase, historical data is collected, preprocessed, and used to train a predictive model. In the real-time phase, anomalies are detected in streaming data in real time, and the model is retrained once a concept drift occurs. In the following, we provide a more detailed explanation of each phase.

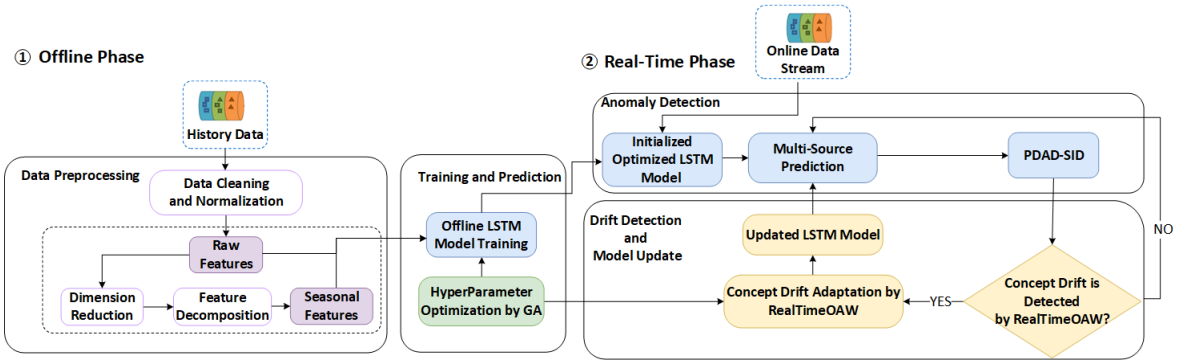


Figure 3.3: Overview of the proposed Scalable and Adaptable Prediction-Driven Anomaly Detection.

3.5.1 Offline Phase

The offline phase comprises two main components of (A) Data Preprocessing, (B) Training and Prediction.

(A) Data Preprocessing: The Data Preprocessing module converts the time-series data into the proper form required to be used in a neural network. First, the data is cleansed from errors (e.g., missing values, outliers) and normalized. Second, we employ the PCA method as our preprocessing module to take into account the correlation between the features [80]. Using PCA, correlated features can be transformed into uncorrelated features called orthogonal features, also known as principal components. Principal components capture the orientations of the data, explaining the highest proportion of variance. The correlations between relevant features are considered during the process of reducing the dimension. We note that the incorporation of dimension reduction can further enhance the efficiency of our proposed method. By leveraging the covariance matrix of data with a high number of dimensions, PCA allows for its projection onto a new space. In this transformed space, the axes are aligned with the eigenvectors of the covariance matrix, prioritized based on the magnitude of their corresponding eigenvalues. This process effectively reduces the data size by retaining only the directions that capture the most informative aspects, i.e., those associated with higher eigenvalues.

Next, the time series of each sample feature is decomposed via time-series decomposition. For feature decomposition, we apply the so-called Time Series Analysis (TSA) decomposition [81] method, which is a procedure for predicting time-series data based on an additive model. More specifically, the obtained seasonal features of the record are concatenated with the selected features of the record by PCA as an additional input before being fed to the Training and Prediction module, to be explained next.

(B) Training and Prediction: The Training module is responsible for training an LSTM model, which can be used in conjunction with the feature decomposition process

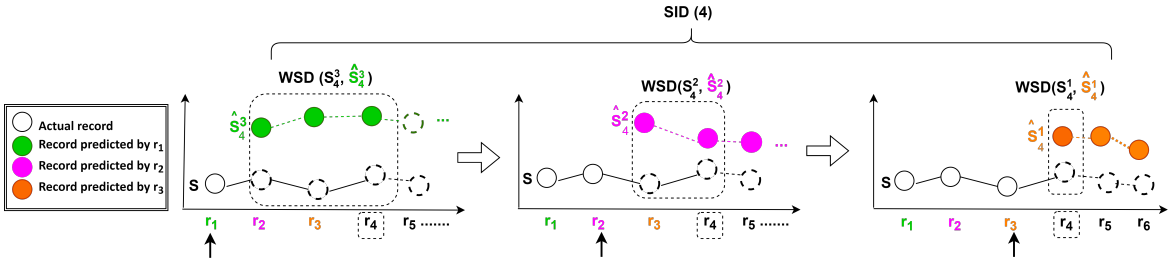


Figure 3.4: An example demonstrating the calculation of SID with the maximum prediction horizon length L=3

explained in Section (A) to make multi-source predictions using a sequence of records. LSTM provides an effective method for analyzing time correlation features of time-series data. Capturing time correlation is critical as anomalies may be related to previous period data. As a result, by analyzing the time correlation present in time-series data, we can effectively detect anomalies. We note, however, that LSTM models inherently fall short of learning complicated seasonal patterns across a given multi-seasonal time-series data. To cope with this, the proposed method explicitly considers the seasonal aspects of the data as inputs to the LSTM model by feature decomposition. Extracting and using the seasonal features not only can help the LSTM model learn complicated seasonal patterns but also lead to more efficient and shorter training. we employ a hyperparameter optimization technique based on GA [82]. This technique involves searching the hyperparameter space using GA to identify the combination of hyperparameters that best fit the data and produce accurate predictions. By leveraging GA-based hyperparameter optimization, we aim to improve the overall performance of our LSTM model. GA identifies optimal hyperparameter values by leveraging information sharing and cooperation among individuals in a population. We employ GA to identify the optimal hyperparameters and LSTM network architecture, including the number of hidden units, training times, gradient threshold, and learning rate, for achieving the best performance [82].

3.5.2 Real-Time Phase

In the real-time phase, the proposed method processes online streams of data generated continuously over time. This phase starts with the utilization of the LSTM model generated during the offline phase to predict the future records of the data stream to compute their anomaly probability utilizing PDAD-SID. PDAD-SID, a multi-source prediction approach, uses a record-to-sequence predictor. This differs from traditional record-to-record predictors [37] by considering a sequence of records for making predictions at different time points rather than only one. Multiple predictions from multiple records form a unified estimated value of the target record, which can be used to determine whether or not it is anomalous, depending on the target record's deviation from the unified estimated value.

Algorithm 3.1 describes the pseudo-code of the proposed PDAD-SID anomaly detection method. Furthermore, it is crucial to update the anomaly detection model frequently to keep up with the changes in data distribution. Therefore, upon detecting concept drift in the new data streams, the LSTM model will undergo retraining to align with the new data. As a result, the proposed method can effectively adjust to constantly evolving traffic data patterns and consistently achieve accurate anomaly detection. The Real-Time phase comprises two main components of (A) Anomaly Detection, and (B) Drift Detection and Model Update.

(A) Anomaly Detection: The real-time anomaly detection module relies on the calculation of two key metrics, namely, Sequence Inconsistency Distance (SID) metric and Anomaly Probability (AP), which are explained below.

SID Metric: Let $\mathbf{r}_i$ and $\mathbf{r}_j$ denote records i and j, respectively, D be the dimension of each record, and $r_i^{(l)}$ be the component l of record i. We obtained the Record Distance (RD) between records i and j as follows:

$$\mathrm{RD}(\mathbf{r}_i, \mathbf{r}_j) = \frac{1}{D}\sum_{l=1}^{D}(r_i^{(l)} - r_j^{(l)})^2. \tag{3.1}$$

Next, let us define sequence $\mathbf{S}_i^N$, $\forall i \geq N$, of length N, which comprises N consecutive records ending at record i:

$$\mathbf{S}_i^N = (\mathbf{r}_{i-N+1}, ..., \mathbf{r}_{i-1}, \mathbf{r}_i). \tag{3.2}$$

In order to obtain a uniform scale for measuring the distance between two sequences, the sequence distance metric should be independent of the sequence length. Thus, we calculate the distance between sequence $\mathbf{S}_i^N$ of actual record values and sequence $\hat{\mathbf{S}}_i^N$ of predicted record values via Weighted Sequence Distance (WSD), which is defined as follows:

$$\mathrm{WSD}\left(\mathbf{S}_i^N, \hat{\mathbf{S}}_i^N\right) = \frac{\sum_{m=1}^{N} e^{(N-m)} \cdot \mathrm{RD}\left(\mathbf{r}_{i-m+1}, \hat{\mathbf{r}}_{i-m+1}\right)}{\sum_{m=1}^{N} e^{(N-m)}}, \tag{3.3}$$

where $\mathbf{r}_{i-m+1}$ and $\hat{\mathbf{r}}_{i-m+1}$ are the actual and predicted values of record $i-m+1$. In the definition of the WSD function, time decay weight $e^{(N-m)}$ is associated with record $i-m+1$ in order to assign a greater level of reference to more recent records.

To quantify the deviation of the target record i from the estimated value, we define a new metric called SID, which is given by:

$$\mathrm{SID}\,(i) = \frac{\sum_{k=1}^{L} P(i-k) \cdot \mathrm{WSD}(\mathbf{S}_i^k, \hat{\mathbf{S}}_i^k)}{\sum_{k=1}^{L} P(i-k)}, \tag{3.4}$$

where L is the prediction horizon, which determines the maximum length of the predicted sequence, and $P(i-k)$ denotes the probability that record $i-k$ is normal. The proposed SID

metric alleviates the impact of anomalous records within the historical data by weighting the predictions (see Eq. (3.4)). Fig. 3.4 provides a visual representation of the calculation of SID in a specific case where $L = 3$. For example, let us assume that we aim to calculate the SID of target 4 for $L = 3$. Once the actual record $\mathbf{r}_1$ arrives, the model predicts the next three records $\mathbf{r}_2$, $\mathbf{r}_3$, and $\mathbf{r}_4$. Similarly, once record $\mathbf{r}_2$ arrives, the model predicts $\mathbf{r}_3$, $\mathbf{r}_4$, and $\mathbf{r}_5$. Likewise, once $\mathbf{r}_3$ arrives, the model predicts $\mathbf{r}_4$, $\mathbf{r}_5$, and $\mathbf{r}_6$. A three-step process is carried out to calculate SID(4) of the target 4. First, we calculate the WSD of the actual and matching predicted sequences up to target 4, i.e., the distances between $\mathbf{S}_4^1$ and $\hat{\mathbf{S}}_4^1$, between $\mathbf{S}_4^2$ and $\hat{\mathbf{S}}_4^2$, and between $\mathbf{S}_4^3$ and $\hat{\mathbf{S}}_4^3$ and then normalize it by $\sum_{m=1}^{N} e^{(N-m)}$, as shown in Eq. (3.3). Subsequently, SID is obtained as the normalized weighted sum of WSDs, see Eq. (3.4).

Anomaly Probability: Given that the range of the SID metric is application-specific, we aim to map the value of SID(i) of target i to its probability AP(i) of being anomalous as follows:

$$\mathrm{AP}(i) = \varphi\left(\frac{1}{1 + e^{-C(\mathrm{SID}(i)-\mu)}}\right), \tag{3.5}$$

where $\varphi(\cdot)$ is the logistic mapping function, μ is the mean of the SIDs of some of the targets, C is the logistic growth rate, which is set to $1/\sigma^2$, where σ is the variance of the SIDs of some of the targets. To determine the optimal C and μ, we use an automated iterative algorithm, which is applied to the training data. As time passes and the probability of the targets becomes more accurate, we calculate μ and C according to the mean and standard deviation of previous SIDs, respectively, until the algorithm converges to stable values of μ and C. Using Eq. (3.5) to map SIDs to APs not only allows for a better distinguishing between normal and anomalous data, but it helps the values of APs become less application-specific. Finally, in order to assign larger weights for normal records in the calculation of SID, we substitute $P(i-k)$ in Eq. (3.4) with $(1 - \mathrm{AP}(i-k))$. Thus, Eq. (3.4) can be re-written as follows:

$$\mathrm{SID}\,(i) = \frac{\sum_{k=1}^{L}(1 - \mathrm{AP}(i-k)) \cdot \mathrm{WSD}(\mathbf{S}_i^k, \hat{\mathbf{S}}_i^k)}{\sum_{k=1}^{L}(1 - \mathrm{AP}(i-k))}. \tag{3.6}$$

Algorithm 3.1 utilizes prediction horizon L, reference series length S, and historical records (r_{i-K} to r_{i-1}) to calculate SID for real-time anomaly detection. Initially, logistic growth rate C and mean μ are set, and anomaly probabilities AP are initialized (lines 1 to 4 of Algorithm 3.1). For each incoming record $\mathbf{r}_i$ in the data stream, the algorithm utilizes the LSTM model generated during the offline phase to predict future records. This involves predicting multiple steps ahead, up to the prediction horizon L (lines 5 to 6 of Algorithm 3.1). For each predicted step (k from 1 to L), the algorithm calculates WSD using Eq. (3.3) (lines 7 in Algorithm 3.1). The SID is then updated using Eq. (3.4), which involves the weighted sum of WSDs. This step provides a measure of the deviation of the target record from the estimated value, considering the historical context and the probability

that previous records were normal. The SID values are mapped to AP using the logistic mapping function described in Eq. (3.5). It refines SID and AP iteratively until convergence and dynamically adjusts the parameters (lines 10 to 13 of Algorithm 3.1). Likewise, the algorithm iteratively updates parameters μ and σ until convergence, allowing it to adapt to changing data patterns (lines 14 to 16 of Algorithm 3.1). The resulting SID values are then updated to assign larger weights for normal records in the calculation of SID, the probability of a record being normal, denoted as $(1-\mathrm{AP}(i-k))$ is substituted in the calculation of SID (line 18 of Algorithm 3.1).

(B) Drift Detection and Model Update: Streaming data may be subject to a variety of data distribution changes, which may occur in dynamic cloud environments. Thus, the anomaly detection model should be updated frequently in order to keep up with the changes in data distribution. To cope with this issue, we propose a RealTimeOAW method to detect the concept drift and adapt to it. Our proposed RealTimeOAW algorithm combines sliding and adaptive window-based methods with performance-based approaches. The RealTimeOAW algorithm is presented in Algorithm 3.3, which detects concept drift within streaming data and subsequently incorporates new records to update the model. Algorithm 3.2 (HyperparameterOptimization Algorithm) uses GA to tune and optimize the hyperparameters of the RealTimeOAW algorithm.

Our proposed RealTimeOAW algorithm exploits two types of windows: (i) sliding window for the detection of the concept drift and (ii) adaptive window for storing newly arrived records. The sliding window has a fixed size of L_s, which contains the most recent records. The adaptive window, which has a maximum size of L_a, can adapt its size dynamically based on the number of arriving records. When a new record arrives, it is added to the adaptive window. If the number of records in the adaptive window exceeds L_a, the oldest records are removed from the window. By keeping a window of recently arrived records, the adaptive window ensures that the algorithm can adapt to the changes in data distribution. The sliding window W_i associated with data record r_i contains L_s records from record $i - L_s$ to record i. The Anomaly Rate (AR) within the window can be calculated by considering both the anomaly probability and the number of anomalous data records in the corresponding window. If the anomaly probability of each record data exceeds a pre-defined threshold T, i.e., $\mathrm{AP}(i) > T$, then record i is considered as an anomalous record. Let W denote the set of the anomaly probabilities in the given sliding window W_i as follows:

$$\mathrm{W} = \{AP(i-L_s), \ldots, AP(i-1), AP(i)\}\,. \tag{3.7}$$

We then define W_A as the set of APs ($\in W$) that are greater than the given threshold T:

$$W_A = \{AP(i) \in W \mid AP(i) > T\}\,. \tag{3.8}$$

Algorithm 3.1: PDAD-SID

Require: Target record r_i, prediction horizon L, reference series length S, records r_{i-K} to r_{i-1}, AP$(i-k)$ for $k \in [1, L]$
Ensure: SID(i)

$C \leftarrow 1, \mu \leftarrow 0.5$
for $i = 0 : s-1$ **do**
  AP$(i) \leftarrow 0$
end for
for $k = 1 : L$ **do**
  Predict $\hat{\mathbf{S}}_i^k$
  Calculate WSD $\left(\mathbf{S}_i^k, \hat{\mathbf{S}}_i^k\right)$ using Eq. (3.3)
end for
while convergence condition is not met **do**
  for $i = L : s-1$ **do**
    Calculate SID(i) using Eq. (3.4)
    Calculate AP(i) using Eq. (3.5)
  end for
  $\mu \leftarrow$ Mean of the SIDs
  $\sigma \leftarrow$ Variance of the SIDs
  $C \leftarrow 1/\sigma^2$
end while
Update SID(i) using Eq. (3.6)

Algorithm 3.2: HyperParameter-sOptimization

Require: Space: configuration space, MaxTime: iteration for hyperparameter search
Ensure: $Optimized_{HP}$: the detected optimal hyperparameter values
MinAR: the average overall AR

$MinAR \leftarrow 1$
for $j = 1 : MaxTime$ **do**
  A_{th},D_{th},L_s,$L_a \leftarrow$ Genetic Algorithm(Space)
  $AR \leftarrow$ RealTimeOAW (R, LSTM, L, s, A_{th},D_{th},L_s,L_a)
  if $MinAR \geq AR$ **then**
    $MinAR \leftarrow AR$
    $Optimized_{HP} \leftarrow A_{th}$,$D_{th}$,$L_s$,$L_a$
  end if
end for
return $MinAR$, $Optimized_{HP}$

Algorithm 3.3: RealTimeOAW

Require: R: Data stream, LSTM: Pre-trained LSTM model, L: Prediction horizon, S: Series length, A_{th}: Alert threshold, D_{th}: Drift threshold, L_s: Fixed sliding window size, L_a: Max adaptive window size
Ensure: AR

$Condition \leftarrow Normal$
$AdaptWin \leftarrow \emptyset$
for r_i in R **do**
  $PDAD - SID(i, L, s)$
  $W_i \leftarrow$ Window with last L_s records
  $ARWin_i \leftarrow AR(W_i)$ {Current window AR}
  $ARWin_{i-L_s} \leftarrow AR(W_{i-L_s})$ {Last window AR}
  if $(Condition = Normal)$ and $(ARWin_i \geq A_{th} * ARWin_{i-L_s})$ **then**
    $AdaptWin \leftarrow AdaptWin \cup r_i$ {collecting new records}
    $Condition \leftarrow Alert$
  else if $(Condition = Alert)$ **then**
    $L'_a \leftarrow Len(AdaptWin)$
    if $ARWin_i \geq D_{th} * ARWin_{i-L_s}$ **then**
      $Condition \leftarrow Drift$
      $j \leftarrow$i {Determine the AR for the first concept drifted window}
      $UpdatedLSTM \leftarrow$retrain LSTM on $AdaptWin$
    else if $(ARWin_i < A_{th} * ARWin_{i-L_s}$ or $L'_a == L_a)$ {False alarm} **then**
      $AdaptWin \leftarrow \emptyset$
      $Condition \leftarrow Normal$
    else
      $AdaptWin \leftarrow$AdaptWin $\cup \{r_i\}$
    end if
  else if $(Condition = Drift)$ **then**
    $L'_a \leftarrow Len(AdaptWin)$
    if $(ARWin_i \geq A_{th} * ARWin_{j+L_s}$ or $L'_a == L_a)$ **then**
      $UpdatedLSTM \leftarrow$retrain UpdatedLSTM on $AdaptWin$
      $AdaptWin \leftarrow \emptyset$
      $Condition \leftarrow Normal$
    else
      $AdaptWin \leftarrow AdaptWin \cup \{r_i\}$
    end if
  end if
end for
return AR

Then, we calculate AR as follows:

$$\text{AR} = \frac{|W_A|}{|W|}. \tag{3.9}$$

To detect the concept drift, we define thresholds A_{th} and D_{th}, which are used to trigger the alert and drift levels, respectively. More specifically, when the difference between the ARs

of the current sliding window i and the sliding window i - L_s exceeds A_{th}, the alert level is activated, prompting the adaptive window to commence the collection of new incoming data records (lines 5 to 10 of Algorithm 3.3). Similarly, a drift is detected once the difference between the ARs of the current sliding window i and the sliding window i - L_s exceeds D_{th}. At this point, the old learner is retrained on the newly collected records in the adaptive window (lines 12 to 16 of Algorithm 3.3). Specifically, the system enters the retraining phase when one of the subsequent criteria is met: If the current state is normal, the system enters the alert state when the AR of the current sliding window exceeds $A_{th} \times ARWin_{i-L_s}$. If the current state is alert, the system enters the drift state when the AR of the current sliding window exceeds $D_{th} \times ARWin_{i-L_s}$. If the AR of the current window is in either the alert state or the drift state and it does not exceed its threshold or the maximum adaptive window size L_a is reached, the system switches back to the normal state and releases the adaptive window. Once the adaptive window is filled with enough new records, the algorithm retrains an LSTM model on the combined data from the adaptive window and the fixed sliding window. After retraining, the system switches back to the normal state, and the adaptive window is released.

To ensure accurate and consistent learning, the adaptive window is designed to continue collecting records until either of the following conditions is met: (1) the new AR exceeds the alert threshold A_{th} in relation to the starting point of the drift, signifying the current learner's inability to handle the new data record and requiring an update, and (2) the adaptive window reaches its maximum capacity of L_a, guaranteeing that real-time constraints are met. The learner is then updated with the most recent records within the adaptive window, resulting in increased resilience and restoring the system to its normal state (lines 23 to 32 of Algorithm 3.3). Conversely, if the sliding window AR ceases to rise or even drops to the normal level during the alert condition, it is considered as a false alarm. Upon the release of the adaptive window, the system reverts to its normal state. This allows for the monitoring of new drift occurrences (lines 18 to 22 of Algorithm 3.3).

In order to achieve optimal performance, the hyperparameters of the LSTM models and RealTimeOAW must be tuned and optimized. Two categories of hyperparameters require tuning in the LSTM models: model design and model training hyperparameters. Model design hyperparameters are set during the model design process and include parameters such as the number of layers, learning rate, and dropout rate. In contrast, model training hyperparameters balance training speed and model performance and include parameters such as batch size and epoch number. It is important to note that these hyperparameters have a direct impact on the structure, effectiveness, and efficiency of the LSTM models. We note that four parameters play an essential role in the RealTimeOAW algorithm, A_{th}, D_{th}, L_s, and L_a. The performance of the RealTimeOAW algorithm is directly influenced by these parameters. In RealTimeOAW, GA is employed to tune these hyperparameters, leading to the creation of an optimal adaptive learner capable of handling both continuous and

Table 3.1: Hyperparameter configuration of LSTM and RealTimeOAW on (A) KDDCup99, (B) IoTID20, and (C) WUSTL-IIoT with Optimal Values.

Model	Hyperparameter	Search Range	A	B	C
LSTM	Number of Epochs	[20, 100]	100	87	100
	Learning Rate	[0.0001, 0.01]	0.001	0.0001	0.001
	Batch Size	[10, 50]	32	45	39
RealTime OAW	A_{th}	(0, 0.1)	0.092	0.089	0.095
	D_{th}	(0, 0.08)	0.03	0.045	0.034
	L_s	[100, 600]	270	150	325
	L_a	[400, 4000]	2500	1235	3240

discrete parameters. The Hyperparameter Optimized Algorithm is shown in Algorithm 3.2, which uses GA to identify the optimal combination of hyperparameters that return the smallest overall AR. The detected optimal hyperparameters are subsequently fed to the RealTimeOAW algorithm to build an optimized model to detect anomalies accurately.

3.6 Performance Evaluation

In this section, we conduct a performance evaluation of our proposed SAPDAD method. After describing our simulation environment, datasets, and evaluation metrics, we present our findings. Finally, we assess the impact of the anomaly probability threshold T on the performance.

3.6.1 Experiment Settings

Our evaluations were conducted on a system equipped with a quad-core CPU (model: Intel Core i7-7700, 3.60 GHz) and 16 GB of RAM. The inference module of our prediction model was implemented using the Tensorflow-addons platform (V.: 0.14), while the decomposition module utilized the TSA Decomposition library (V.: 0.4). We conducted a hold-out validation approach, where the initial model training utilized the first 10% of the data, while the remaining 90% was reserved for online testing purposes. The LSTM network structure used for each dataset is based on $[\#RawFeatures + \#SeasonalFeatures, TimeStep, \#RawFeatures \times L]$. Three different time steps were evaluated, where the seasonality length was 24 (daily seasonality length), 72, or 168 (weekly seasonality length). Dropout was disabled. In order to avoid over-fitting, a weight decay of 6×10^{-6} was applied during the training.

Table 3.1 presents the hyperparameters of LSTMs and RealTimeOAWs, which were automatically tuned using GA. The experimental setup consisted of a population size of 70, a crossover rate of 0.7, and a mutation rate of 0.15. Further, we determined the optimal

threshold for each of the time-series datasets, considering their common properties and validation data. We consider a set of threshold values for anomaly probability, including $T = \{0.55, 0.60, 0.65, 0.70, 0.75\}$. Our simulation results indicate that the optimal threshold value falls within the range of 0.55 to 0.75 because the large thresholds result in more false anomalies, whereas the small thresholds prolong the execution time. The values of prediction horizon L are selected from $\{10, 15, 20, 30\}$.

3.6.1.1 Description of Datasets:

Our evaluations are carried out on three high-dimensional anomaly detection datasets, namely, the KDDCup99, IoTID20, and WUSTL-IIoT.

- KDDCup99 [41] is an imbalanced intrusion detection dataset, which includes a variety of hand-injected anomalies (i.e., network attacks) into the normal network data. The dataset has 43 dimensions, with an anomaly rate of 1.77%.
- IoTID20 [83] is an IoT traffic dataset for anomaly detection where the distribution of records is unbalanced, with 94% categorized as normal and the remaining 6% labeled as anomalous. The dataset was generated by employing virtual machines representing both normal network behavior and attack scenarios, mimicking IoT services through the utilization of the node-red tool. Subsequently, features were extracted using the Information Security Center of Excellence (ISCX) flow meter program. The reduced IoTID20 dataset used in this work contains 6253 records, which were randomly sampled based on the time slot at a rate of one data record per 10 time slots.
- WUSTL-IIoT [84] is a network data of IIoT and it is collected from real-world industrial systems. This dataset consists of a variety of IIoT components such as sensors, actuators, Human-Machine Interfaces (HMI), Programmable Logic Controllers (PLC), data loggers, and alarm systems, all aimed at mimicking actual industrial operations. The dataset is characterized by 41 distinct features, chosen for their variability during different attack phases. The types of cyber attacks represented in this dataset include command injection, Denial of Service (DoS), reconnaissance, and the use of backdoors.

3.6.1.2 Evaluation Metrics

In the following, we present our evaluation metrics. First, we evaluate the effectiveness of different detection methods under study by the so-called Area Under Curve (AUC), which measures the accuracy. A large value of AUC indicates that the model has a better measure of separability, which is key for anomaly detection to meet our accuracy and drift adaptive requirements. Second, we consider two Quality of Service (QoS) parameters related to ML and data analytics, namely, average execution time and processing rate. Average execution

time is the time required for processing a single data record, including anomaly detection, drift detection, and model updating. A short execution time is crucial to meet the real-time and time-efficiency requirements of the anomaly detection system. Moreover, the processing rate measures the number of processed data records per second. A large processing rate is essential for scaling the proposed method to a large number of data records, i.e., scalability requirement.

3.6.2 Evaluation Results

We compare the performance of our proposed SAPDAD anomaly detection algorithm with several benchmarks, namely, ORUNADA [40], ASTREAM [44], Online RNN-AD [45], and PDAD-SID [1].

Table 3.2 shows a comparison between our proposed SAPDAD anomaly detection algorithm and our benchmarks for three datasets. We observe from the table that the proposed method outperforms all other methods in terms of AUC, achieving a score of 89.71% on the KDDCup99 dataset, 80.66% on the IoTID20 dataset, and 83.08% on the WUSTL-IIoT dataset. In comparison, ORUNADA [40], ASTREAM [44], Online RNN-AD [45] have a significantly lower AUC scores of 78.83%, 88.11%, and 77.73% on the KDDCup99 dataset, 76.72%, 78.91%, and 74.03% on the IoTID20 dataset, and 77.19%, 78.68%, and 76.28% on the WUSTL-IIoT dataset, respectively. The high AUC achieved by the proposed SAPDAD demonstrates its robustness and also reflects its ability to adapt to evolving data patterns. Next, AUC vs. the number of records is shown in Fig. 3.5, where we observe that as the data volume increases, the proposed SAPDAD method maintains a consistent AUC, indicating its resilience to concept drift. In contrast, other methods such as Online RNN-AD [45], while efficient in processing rate, show variations in their AUC performances. According to Fig. 3.5a, although a small drift occurred in the KDDCup99 dataset at the early stage of the experiment, all methods were able to adjust to it, but at different rates. Our proposed method adapted to all drifts detected at data records 1201, 10854, and 19245, achieving the highest AUC of 89.71%, while the PDAD-SID model's AUC dropped to only 82.15% without drift adaptation. Similarly, in the IoTID20 dataset, as illustrated in Fig. 3.5b, our proposed method achieved an AUC of 80.66% showcasing its ability to adapt to a subtle concept drift detected at data record 1023. In comparison, the PDAD-SID model [1] had a slightly lower AUC of 79.28% without drift adaptation. As illustrated in Fig. 3.5c, our proposed method achieves an AUC of 83.08% showcasing its ability to adapt to a concept drift detected at data records 3380, 9000, and 12584. In comparison, the PDAD-SID method [1] has a slightly lower AUC of 80.31% without any drift adaptation. Therefore, while the PDAD-SID model [1] shows a comparable efficiency in terms of execution time, its lower AUC and processing rate suggest a trade-off between speed and accuracy. This trade-off is evident in Fig. 3.5, where the performance of PDAD-SID falls below SAPDAD, especially for the IoTID20 and WUSTL-IIoT datasets, which

Table 3.2: Performance comparison of different anomaly detection methods with drift adaptation.

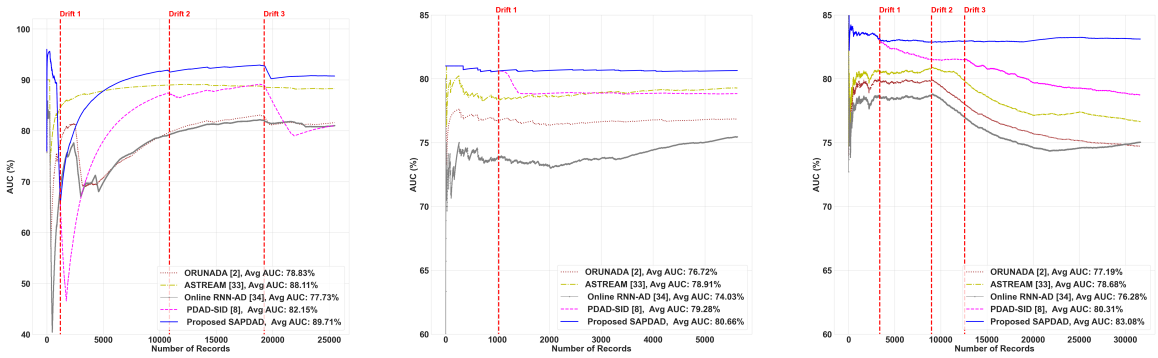
Methods	KDDCup99			IoTID20			WUSTL-IIoT		
	AUC (%)	Avg. Exec. Time (ms)	Proc. Rate (Rec/Sec)	AUC (%)	Avg. Exec. Time (ms)	Proc. Rate (Rec/Sec)	AUC (%)	Avg. Exec. Time (ms)	Proc. Rate (Rec/Sec)
ORUNADA [40]	78.83	2.3491	427	76.72	1.2369	917	77.19	1.9861	501
ASTREAM [44]	88.11	1.8870	531	78.91	1.0670	1117	78.68	2.2210	450
Online RNN-AD [45]	77.73	3.2433	308	74.03	3.0122	332	76.28	4.2452	235
PDAD-SID [1]	82.15	0.0305	2011	79.28	0.0119	2118	80.31	0.0652	1995
Proposed SAPDAD	89.71	0.0525	1945	80.66	0.0299	2021	83.08	0.0801	1984

present more complex and unbalanced data characteristics.

Table 3.2 also presents the average execution time performance of different algorithms under consideration. The proposed model outperforms ORUNADA [40], ASTREAM [44], and Online RNN-AD [45] with significantly shorter average execution times of only 0.0525 ms, 0.0299 ms, and 0.0801 ms, respectively, across the three datasets. The reason for this is mainly due to PDAD-SID's efficiency and sliding window strategy. In terms of record processing rate, ORUNADA [40], ASTREAM [44], and Online RNN-AD [45] outperform the proposed model. However, their AUC falls significantly short compared to the performance achieved by our proposed method. Additionally, in contrast to PDAD-SID, the proposed method demonstrates a significantly improved processing rate of 289, 378, and 302 records per second in the three datasets. While PDAD-SID exhibits shorter execution times for each record compared to the proposed model, its AUC is notably lower than the proposed method. Hence, based on our findings, setting the anomaly probability threshold T to 0.65, 0.70, and 0.70, and the prediction horizon L to 10, 15, and 15 for KDDCup99, IoTID20, and WUSTL-IIoT, respectively lead to a suitable accuracy-efficiency trade-off for practical scenarios (see Table 3.2).

The combination of high AUC scores and the graphical trends in Fig. 3.5 underline SAPDAD's superior performance in adapting to concept drifts, a critical aspect in dynamic environments. The proposed SAPDAD not only achieves a high accuracy (as indicated by AUC) but also maintains efficiency in terms of execution time, thus making it highly suitable for real-time applications. The consistent performance of the proposed method across various datasets, KDDCup99, IoTID20, and WUSTL-IIoT, highlights its robustness and applicability in a wide range of cloud environments.

In addition to high accuracy, the SAPDAD method demonstrates notable efficiency in processing data streams. This efficiency is primarily achieved through its unique architectural design and optimization. Specifically, the use of a lightweight LSTM network structure coupled with a strategic hold-out validation approach ensures swift data processing without compromising the ability to learn from complex data patterns. Furthermore, the application of weight decay during training helps prevent overfitting, thereby maintaining agility in processing new data. Moreover, the results demonstrate operational efficiency, which is essential to run real-time anomaly detection in resource-constrained environments. This is evidenced by the obtained execution times shown in Table 3.2. Additionally, by



(a) (b) (c)

Figure 3.5: AUC of different anomaly detection methods with drift adaptation vs. the number of records for (a) KDDCup99, (b) IoTID20, and (c) WUSTL-IIoT datasets.

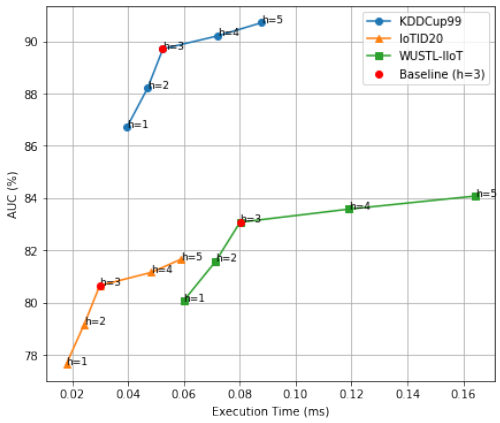


Figure 3.6: AUC vs. execution time for different values of the number h of LSTM hidden layers.

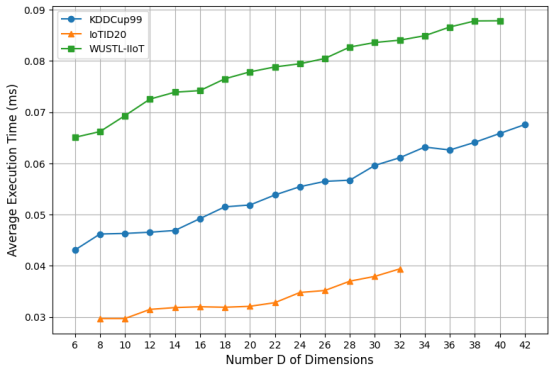
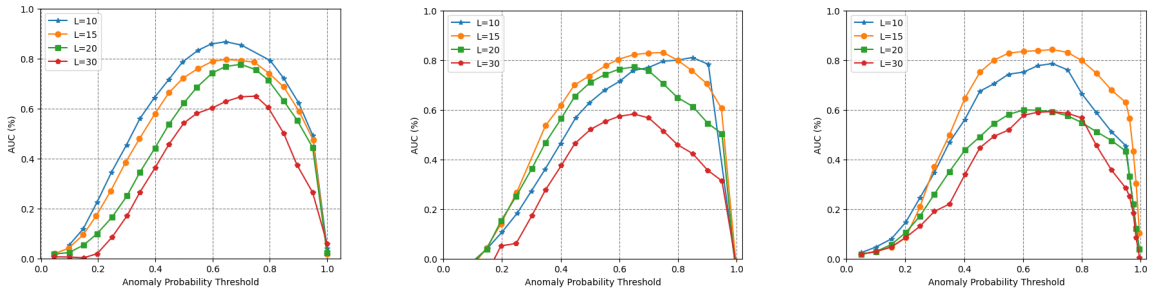


Figure 3.7: Average execution vs. number D of dimensions.

dynamically adjusting to concept drifts and evolving data patterns, the proposed SAPDAD method reduces the need for frequent retraining and/or manual intervention, thereby saving computational resources and time.

Next, we assess the correlation between accuracy and execution time. Fig. 3.6 depicts AUC vs. execution time for different values of the number h of LSTM hidden layers for KDDCup99, IoTID20, and WUSTL-IIoT datasets. We observe from the figure that the AUC exhibits a positive correlation with the number h of hidden layers, which was expected. More specifically, as h increases, the obtained AUC also increases, which comes at the expense of a larger execution time. The increased execution time is a direct consequence of incurring complexity to the underlying LSTM model. Fig. 3.6 is key to help the decision-makers make a trade-off between accuracy and cost.

Next, we examine the trade-off between AUC and execution time for different values



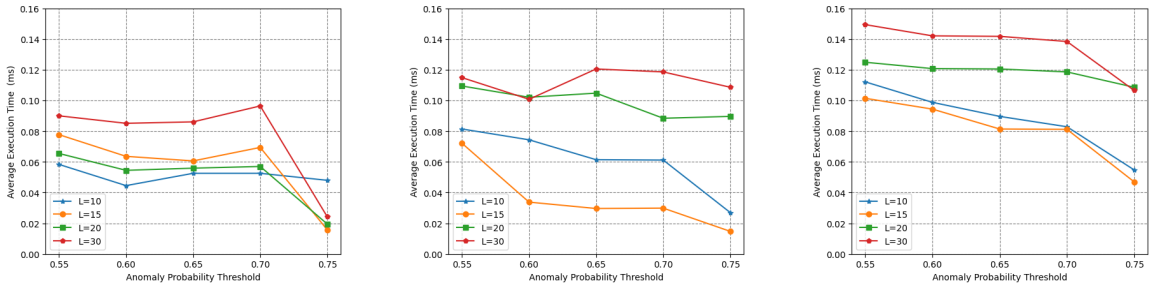
(a) (b) (c)

Figure 3.8: Average AUC vs. anomaly probability threshold T for different values of prediction horizon L in (a) KDDCup99, (b) IoTID20, and (c) WUSTL-IIoT datasets.

of input parameters of anomaly probability threshold T, prediction horizon L, and input dimension D.

(A) AUC: Fig. 3.8 illustrates the AUC of the proposed SAPDAD algorithm for different values of anomaly probability threshold T and prediction horizon L. Given that our proposed algorithm carries out a probabilistic detection rather than classification, we use AUC as our metric, which offers the advantage of being threshold-independent. The values of anomaly probability threshold T and prediction horizon L was selected from $\{0.55, 0.60, 0.65, 0.70, 0.75\}$ and $\{10, 15, 20, 30\}$. As shown in Fig. 3.8a, the highest AUC was achieved for L=10 for a wide range of anomaly probability threshold values. Therefore, we set the prediction horizon L to 10 in the rest of our evaluations. For a fixed prediction horizon L, the optimal anomaly probability threshold T was 0.65 for the KDDCup99 dataset because it resulted in improved AUC. Hence, we consider the data records with an anomaly probability of greater than 0.65 as anomalous. For the IoTID20 dataset, as shown in Fig. 3.8b, L=15 leads to the highest AUC with an anomaly probability threshold T of 0.70. For the WUSTL-IIoT dataset, as shown in Fig. 3.8c, L=15 leads to the highest AUC with an anomaly probability threshold T of 0.70. We note that a higher anomaly probability threshold results in fewer false anomalies, whereas a lower threshold increases the false negatives. Therefore, we determined the anomaly probability threshold T to 0.65, 0.70, and 0.70 in the KDDCup99, IoTID20, and WUSTL-IIoT datasets, respectively.

(B) Execution Time: Fig. 3.9 illustrates the execution time for different values of anomaly probability threshold T and prediction horizon L. According to Fig. 3.9a, Fig. 3.9b, and Fig. 3.9c, it is evident that the average execution time of the proposed SAPDAD decreases as the anomaly probability threshold T increases for a given prediction horizon L. This is due to the fact that the frequency of model updates decreases as the anomaly probability threshold T increases. For a fixed anomaly probability threshold T, the average execution time grows as the prediction horizon L increases. This is due to the



(a) (b) (c)

Figure 3.9: Average execution time vs. anomaly probability threshold T for different values of prediction horizon L in (a) KDDCup99, (b) IoTID20, and (c) WUSTL-IIoT datasets.

fact that increasing the prediction horizon L makes it more difficult to satisfy the criteria for model updates.

(C) Scalability: Next, we examine the scalability of the proposed algorithm. To ensure a fair comparison, Fig. 3.7 illustrates only the results of SAPDAD for different dimensions. The reason is that the execution time of the existing methods was well above a millisecond, even for a small number of dimensions. In the existing methods, as the number of dimensions increased, the average execution time increased significantly. We vary the number D of dimensions from 6 to 42, 8 to 32, and 6 to 40 for KDDCup99, IoTID20, and WUSTL-IIoT datasets, respectively. Among the existing methods tested on the KDDCup99 dataset, the proposed SAPDAD method demonstrates the lowest execution time. More specifically, the obtained execution time is 1.2 ms and 3.6 ms for D=14 and D=42, respectively. Similarly, when applied to the IoTID20 dataset, the proposed SAPDAD method exhibits the lowest execution time of 1 ms and 2.8 ms for D=6 and D=32, respectively. For the WUSTL-IIoT dataset, the proposed SAPDAD method exhibits the lowest execution time of 1.6 ms and 3.1 ms for D=24 and D=40, respectively. We observe from Fig. 3.7 that the execution time of the proposed SAPDAD method grows linearly with the number of dimensions in the three datasets. The small execution time in the IoTID20 dataset compared to the KDDCup99 and WUSTL-IIoT datasets can be attributed to two factors. First, the network traffic patterns in the IoTID20 dataset have less variability and fluctuation, allowing for faster processing during testing. Second, given that there existed a smaller number of concept drifts in the IoTID20 dataset, fewer model adaptations were required during the testing.

Table 3.3: Contributors to the latency (ms) in the proposed SAPDAD.

phase	Module	Latency (ms)		
		KDDCup99	IoTID20	WUSTL-IIoT
Offline	Preprocessing	0.091	0.050	0.12
	Training and Prediction	3.110	2.544	3.857
Real-Time	Anomaly Detection	0.011	0.012	0.014
	Drift Detection and Model Update	0.041	0.017	0.065

Table 3.4: Ablation experiment results for drift adaptation and dimension reduction on AUC in (A) KDDCup99, (B) IoTID20, and (C) WUSTL-IIoT.

Variant	Drift Adaptation	Dimension Reduction	AUC (%)		
			A	B	C
Full System	Yes	Yes	89.71	80.66	83.08
No Drift Adaptation	No	Yes	85.75	76.64	79.10
No Dimension Reduction	Yes	No	87.75	78.64	82.10
No Modules	No	No	82.75	73.64	77.10

3.6.2.1 Latency of Various Modules in SAPDAD

In order to identify the key factors contributing to latency in the proposed SAPDAD, a detailed breakdown of the latency for various modules is presented in Table 3.3. This table provides a comprehensive view of how each phase and module of the approach contributes to the overall latency across KDDCup99, IoTID20, and WUSTL-IIoT datasets. We observe from Table 3.3 that the Training and Prediction module is the primary contributor to the overall latency. This is particularly noticeable in the offline phase, where the latency of this module is significantly higher than the others, recording 3.110 ms for KDDCup99, 2.544 ms for IoTID20, and 3.857 ms for WUSTL-IIoT. The Preprocessing module, which is also in the offline phase, shows a latency of $<$0.1 ms for various datasets. This indicates a relatively minor impact on the overall latency, owing to the preprocessing algorithms' efficiency. During the real-time phase, both Anomaly Detection and Drift Detection and Model Update modules demonstrate low latency values. The Anomaly Detection module records latencies of only 0.011 ms, 0.012 ms, and 0.014 ms for KDDCup99, IoTID20, and WUSTL-IIoT, respectively, reflecting its low computational complexity and efficient operation. Similarly, the Drift Detection and Model Update module shows latencies of 0.041 ms, 0.017 ms, and 0.065 ms for KDDCup99, IoTID20, and WUSTL-IIoT, respectively.

Table 3.5: Comparative analysis of SAPDAD model performance using LSTM, CNN, GRU, and LSTM-CNN architectures for WUSTL-IIoT.

Model	AUC (%)	Execution Time (ms)
SAPDAD with LSTM	83.08	0.0801
SAPDAD with CNN	80.10	0.0676
SAPDAD with GRU	79.95	0.0942
SAPDAD with LSTM-CNN	76.41	0.1452

3.6.2.2 Ablation Study

In the following, we run an ablation evaluation to examine the contribution of different modules of the proposed SAPDAD to the overall performance.

(A) Impact of drift adaptation and dimension reduction: Table 3.4 presents the impact of drift adaptation and dimension reduction modules of SAPDAD on the AUC performance across various datasets. The experiments involve four configurations: (i) full system with both modules enabled, (ii) system without drift adaptation, (iii) system without dimension reduction, and (iv) baseline system without any drift adaptation or dimension reduction. In the full system configuration, the obtained AUCs are the highest among all configurations, reaching 89.71% for KDDCup99, 80.66% for IoTID20, and 83.08% for WUSTL-IIoT, which underscore the combined effectiveness of these modules in enhancing anomaly detection accuracy. When there is no drift adaptation, we observe a noticeable decline in AUC (85.75% for KDDCup99, 76.64% for IoTID20, and 79.10% for WUSTL-IIoT), which highlights the importance of this feature in adapting to changes in data over time. Similarly, the absence of dimension reduction results in decreased AUCs of 87.75%, 78.64%, and 82.10% for the respective datasets, indicating that managing the complexity of the feature space is crucial for maintaining high detection accuracy. The most significant impact is observed in the baseline configuration without any drift adaptation or dimension reduction, where the AUC drops to 82.75%, 73.64%, and 77.10% for KDDCup99, IoTID20, and WUSTL-IIOT datasets, respectively.

3.6.2.3 Performance comparison of various architectures:

Finally, we evaluate the accuracy of the trained prediction model of SAPDAD, which was initially implemented using LSTM networks. To explore the effectiveness of different neural network architectures, we replaced the LSTM network with three alternative configurations: CNN, GRU, and a hybrid network combining LSTM and CNN (LSTM-CNN). As shown in Table 3.5, the LSTM configuration achieved the highest AUC of 83.08% with an execution time of 0.0801 ms, demonstrating its strong predictive capabilities while maintaining computational efficiency. The CNN model, with an AUC of 80.10%, offers a balance between accuracy and efficiency, achieving the lowest execution time of 0.0676 ms, making it a

suitable option for real-time applications with strict latency constraints. The GRU variant, while maintaining a comparable AUC of 79.95%, exhibits a slightly higher execution time of 0.0942 ms. This aligns with expectations, as GRU is typically more computationally efficient than LSTM but still incurs additional processing time compared to CNN. In contrast, the LSTM-CNN hybrid, which integrates both sequential and spatial feature extraction, shows a lower AUC of 76.41% and an execution time of 0.1452 ms. While this configuration provides enhanced feature extraction, the increased computational cost suggests that the combined architecture may introduce additional overhead that does not necessarily translate into improved predictive performance for this dataset.

3.7 Conclusion

In this chapter, we propose an adaptive model for real-time anomaly detection, which is accurate and time-efficient. The proposed SAPDAD method combines a novel drift-handling algorithm (RealTimeOAW), a prediction-driven algorithm (PDAD-SID), and a hyperparameter method (GA) to dynamically adapt to the continuous changes in data streams. The proposed SAPDAD method offers a number of advantages for anomaly detection, including real-time processing capabilities, scalability, adaptability to data pattern changes, and improved accuracy with its prediction-driven strategy, which make it an effective solution for handling the dynamic and complex data streams in real-time cloud environments. The evaluation of the proposed method was conducted using three real-world anomaly detection datasets, namely, KDDCup99, IoTID20, and WUSTL-IIoT. Our trace-driven evaluations demonstrate that the proposed algorithm not only outperforms the existing benchmarks in terms of accuracy but can also detect anomalies in a bounded time.

Chapter 4

Distributed Traces in ML-based Anomaly Detection[1]

4.1 Introduction

The growing adoption of microservices in cloud environments has introduced new challenges for anomaly detection, particularly in real-time cloud environments where services operate with strict latency constraints and dynamic workloads. Unlike monolithic systems, microservices are highly distributed, often relying on interconnected service instances, containerized deployments, and auto-scaling mechanisms. These factors make it difficult to diagnose performance issues, detect anomalies, and ensure system reliability in real time. A major challenge in microservice anomaly detection stems from the complexity of distributed traces—the execution paths of service requests across multiple microservices. These traces are essential for performance diagnosis but are difficult to analyze due to frequent updates, evolving dependencies, and loosely coupled architectures. Moreover, existing anomaly detection methods struggle to handle distributed trace data, high data volumes, and the need for low-latency processing in real-time cloud environments [85].

To address these challenges, this chapter proposes Trace-Driven Anomaly Detection (TDAD) and an Asynchronous Real-Time Federated Learning (ART-FL) framework. TDAD leverages a Graph Neural Network (GNN) to learn vector representations of traces and employs Positive and Unlabeled (PU) learning to train an anomaly detection model with partially labeled data. Building on this, ART-FL enables continuous computation and communication by allowing global model updates without waiting for lagging clients. It integrates logs and traces to capture intra-service behaviors, providing a holistic view of

[1]This chapter is based on two published papers: [3] Mahsa Raeiszadeh, A. Ebrahimzadeh, A. Saleem, R. H. Glitho, J. Eker, and R. A. Mini, "Real-time anomaly detection using distributed tracing in microservice cloud applications," in Proc. *IEEE Cloud Networking (CloudNet)*, pp. 36–44, 2023, and [4] Mahsa Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "Asynchronous Real-Time Federated Learning for Anomaly Detection in Microservice Cloud Applications,"*IEEE Transactions on Machine Learning in Communications and Networking (TMLCN), vol. 3, no. 6, pp. 176 - 194, 2025.*

microservice anomalies. Furthermore, ART-FL incorporates dynamic learning techniques to optimize local and global model convergence, ensuring efficient and timely anomaly detection.

The rest of this chapter is organized as follows. First, we present an illustrative use case followed by the system model and our proposed method and its components. Next, we detail the evaluation process and compare our approach with state-of-the-art methods. Finally, we summarize the key findings and contributions.

4.2 Illustrative Use case

To illustrate the challenges of anomaly detection in distributed traces within real-time cloud, we consider the TrainTicket booking system [85], a cloud-native application built using microservices and containerized environments (Fig. 4.1). This system enables end-users to interact with various services, such as train schedules, ticket booking, and payment processing. Anomalies in the system can manifest as service disruptions, degraded user experience, or security issues, originating from misconfigurations, resource bottlenecks, configuration errors, or communication failures between services. Anomaly detection models aim to identify these anomalies promptly to enable preventive or corrective actions.

This use case highlights the challenges of real-time cloud. While not all services in TrainTicket require real-time processing, critical components such as ticket reservations, payment processing, and live seat availability demand low-latency execution to maintain service reliability and meet user expectations. Any anomaly—whether a performance bottleneck, misconfiguration, or inter-service failure—risks violating SLAs, degrading user experience, and causing financial losses. The TrainTicket use case exemplifies the challenges of anomaly detection in real-time cloud-native applications. Microservices—essential for scalability, modularity, and distributed execution—also introduce complexity due to asynchronous communication, high inter-service dependencies, and dynamic workload variations. These challenges are further amplified in real-time cloud environments, where strict latency constraints demand timely anomaly detection and mitigation.

As shown in Fig. 4.2a, a trace is an array of spans arranged in a tree-like hierarchy, where each span represents a service invocation. Traces and spans are uniquely identified by their IDs. A span records information about the caller and callee, including an operation name that specifies the action performed, such as “GET /api/v1/auth/login" in a REST call. Except for the root, each span is linked to a parent span that initiates its invocation. For example, Span 001 triggers two parallel synchronous calls to Spans 002 and 003, as illustrated in Fig. 4.2a. Fig. 4.2b depicts the timeline of the spans (shown in Fig. 4.2a). Spans 002 and 003, both originating from Span 001, are executed in parallel, leading to a temporal overlap in their execution timelines. Meanwhile, Span 005, an asynchronous invocation initiated by Span 003, is completed after the completion of Span 003.

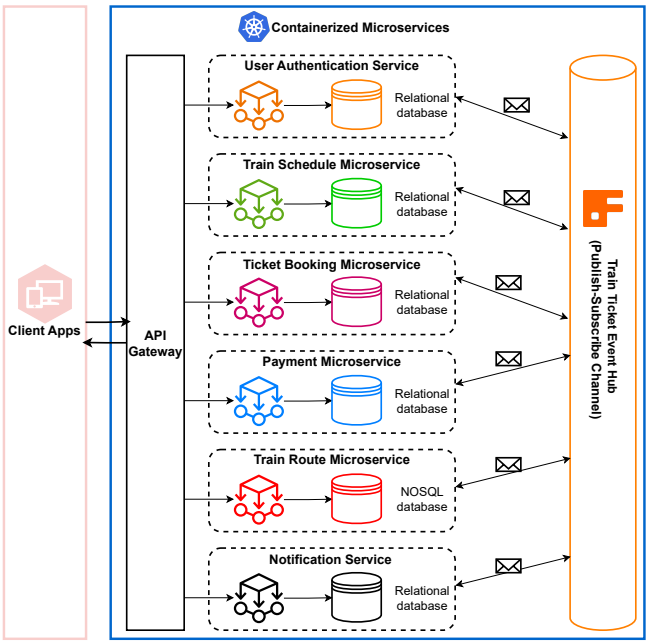
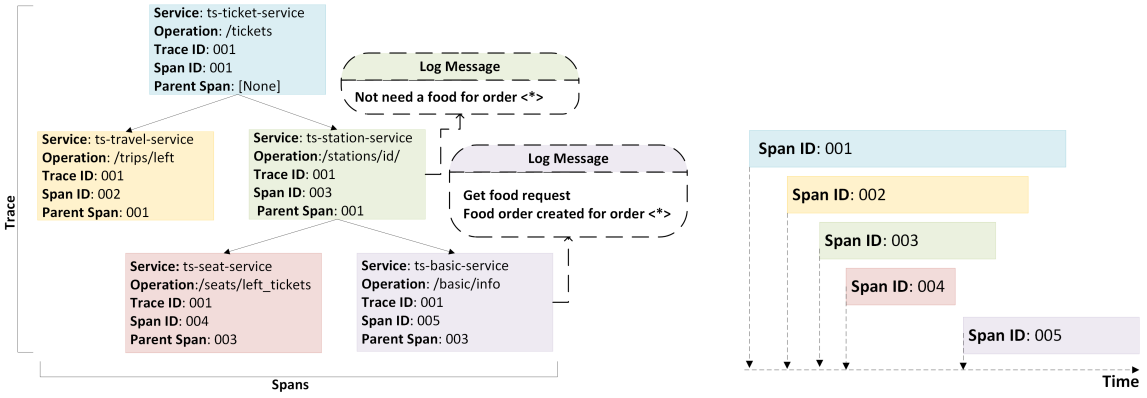


Figure 4.1: Illustrative use case of a TrainTicket microservice cloud application



(a) (b)

Figure 4.2: (a) Illustration of trace, spans, and logs within the TrianTicket microservice application (b) Timeline of Spans in (a).

Current trace anomaly detection methods [50, 51, 52] view traces as service interaction sequences, and log anomaly detection methods [61, 69, 70] treat them as log event series. However, applying these methods effectively in microservices is challenging due to several reasons. First, detecting anomalies requires aggregating logs from multiple services, but existing trace methods overlook log data, limiting their detection capabilities to anomalies within trace structures alone. Fig. 4.2a highlights a log-level anomaly where Span 003's logs indicate no food request for a train ticket, yet Span 005's logs show a food order for that ticket. Detecting such anomalies necessitates merging logs from both Spans 003 and 005.

Second, traces have complex structures, including hierarchical, parallel, and asynchronous calls, which are not fully captured by sequential representations. This simplification misses the causal relationships and the temporal dynamics within and across spans. For instance, in the sequence-based representation, two adjacent log events within Span 001 may appear distant due to the insertion of log events from Span 001's descendant spans. Furthermore, traces often involve parallel or asynchronous service calls. Figs. 4.2a and 4.2b show a trace with an asynchronous (Span 005) and two parallel calls (Spans 002 and 003). This setup leads to the interleaving of log events from Spans 002, 003, and their children, as well as between Span 005 and Span 003, in varied sequences. Merging these events into a single chronological sequence obscures the unique aspects of parallel and asynchronous calls, emphasizing the need to maintain trace structure in log integration.

Although traces are commonly used in anomaly detection [50, 51, 52], they are found to be inadequate for identifying all potential anomalies. They primarily provide a broad view of the system and record interactions across microservices. This high-level data allows for basic, coarse-grained queries but lacks detailed intra-service data. For example, granular details like memory usage, critical for assessing the internal state of a service, are absent in trace data. This limitation underlines the fact that network-related issues, such as latency, cannot wholly encapsulate anomalies associated with resource constraints.

4.3 System Model

In large-scale industrial microservice systems, thousands of microservices across distributed machines generate billions of runtime data points daily, including traces and logs. Our system model involves two main entities, (i) participants, who act as clients, and (ii) a cloud server. Let $\mathcal{K} = \{1, \ldots, K\}$ represent the set of K clients, where each client has a distinct dataset $\mathcal{D}_{k \in \mathcal{K}}$ and represents $n_k = |\mathcal{D}_k|$ to be the number of samples on client k. We denote $N = \sum_{k=1}^{K} |\mathcal{D}_k|$ as the total number of samples in K clients. Each client k independently trains a local model $\mathbf{w}_k$ using its dataset $\mathcal{D}_k$. Notably, only the local model parameters are transmitted to the cloud server. This process deviates from traditional centralized training, where datasets are aggregated, $\mathbf{D} = \cup_{k \in \mathcal{K}} \mathcal{D}_k$, before model training. The local models, represented by $\mathbf{w}_k$, collectively contribute to the generation of a global model $\mathbf{w}_G$ through aggregation at the server. In this system, clients act as collaborative participants, working together to train an ML model, which is coordinated by an aggregate server. The symbols used in the following sections are summarized in Table 4.1 for reference.

4.4 Problem Formulation

In a large-scale system comprising M microservices, each microservice aggregates logs and traces separately. As previously discussed, there are K clients, each running an application

Table 4.1: Summary of main notations.

Symbol	Definition
$\mathbf{w}_G$	Global model parameters
$\mathbf{w}_k$	Local model parameters for client k
π_p	Class prior probability of anomalous traces
T_z	Candidate threshold for anomaly detection
α_{ij}	Attention coefficient from node j to i in the graph
σ	Standard deviation
N	Total number of samples
$\ell_i(x_i, y_i; \mathbf{w}_k)$	Loss function for the data point
η_t	Learning rate at iteration t
λ	Regularization parameter
Γ	Decay coefficient
T_e^0	Initial threshold for anomaly detection
$F(\mathbf{w}_G)$	Global objective function
ξ_k	Gradients computed during local updates
r_t	Multiplier for dynamic learning rate
y_i	Output of the MLP for a given instance i.
q_i	Anomaly probability for instance i.
T_e	Anomaly detection threshold.
T_z'	Updated candidate threshold in refinement.
d	Step size for candidate threshold generation.
C_x	Cluster assigned by K-means.
$\mathcal{Q}$	Set of all anomaly probabilities.
$\mathcal{F}$	K-means clustering function.
$\mathcal{C}$	Set of all clusters.
$\mathcal{Q}_A$	Subset of anomaly probabilities above the threshold.
$\mathcal{Q}_N$	Subset of anomaly probabilities below the threshold.

with M microservices. This means that there are multiple instances of the application, each managed by a different client. In an observation window of length T (where data collected in this window forms a sample), we define multi-source data as $\mathbf{X} = \left\{\left(\mathbf{X}_m^{\mathcal{L}}, \mathbf{X}_m^{\mathcal{T}}\right)\right\}_{m=1}^{M}$, where $\mathcal{L}$ represents the collection of log events from all microservices. Therefore, $\mathbf{X}_m^{\mathcal{L}}$ is specific to the microservice m and denotes the log events associated with it. $\mathcal{T}$ indicates the collection of trace records from all microservices. Therefore, $\mathbf{X}_m^{\mathcal{T}}$ represents the trace records for the microservice m during the observation window T.

Let us define $\mathbf{w}_k$ as the local model for client k. We assume that for any $k \neq k'$, we have $\mathcal{D}_k \cap \mathcal{D}_{k'} = \emptyset$. The local empirical loss for client k is then given by:

$$f_k(\mathbf{w}_k) \stackrel{\text{def}}{=} \frac{1}{n_k} \sum_{i \in \mathcal{D}_k} \ell_i(x_i, y_i; \mathbf{w}_k), \tag{4.1}$$

where $\mathbf{w}_k$ is the local model parameter and $\ell_i(x_i, y_i; \mathbf{w}_k)$ represents the loss function for the

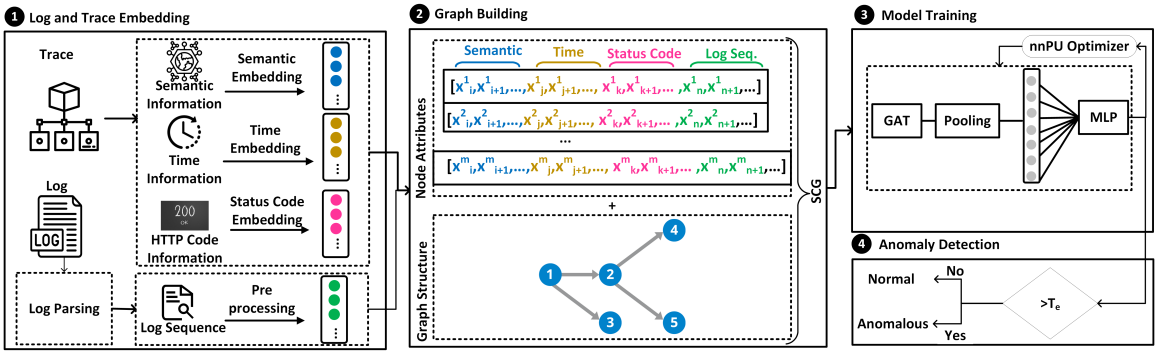


Figure 4.3: Overview of the local model method for the edge clients.

data point $\{x_i, y_i\}$. The global objective function is obtained as follows:

$$F(\mathbf{w}_G) = \sum_{k=1}^{K} \frac{n_k}{N} f_k(\mathbf{w}_G), \tag{4.2}$$

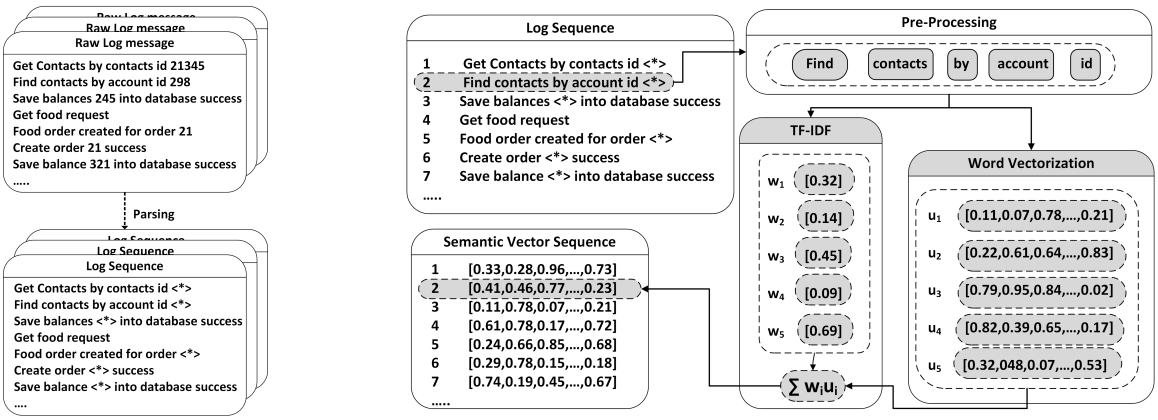
where $\mathbf{w_G}$ is the aggregated global model. The overall goal is to find the model $\mathbf{w}_G^*$ that minimizes $F(\mathbf{w}_G)$:

$$\mathbf{w}_G^* = \arg\min F(\mathbf{w}_G). \tag{4.3}$$

With these considerations in mind, the goal at the local client is to solve the problem of determining whether $\mathbf{X}_{[1:M]}$ is an anomaly or not in real time.

4.5 Proposed Solution

Fig. 4.3 illustrates an overview of our proposed method, which is called Microservice Federated Learning Anomaly Detection (MS-FLAD). Our proposed MS-FLAD method comprises four main components: (i) Log and Trace Embedding, (ii) Graph Building, (iii) Model Training, and (iv) Anomaly Detection. First, log parsing involves processing the incoming logs to extract log events from the messages they contain. Then, a vector representation, including semantic information, is generated for each span through the process of Span Embedding. Second, Graph Building creates an SCG for each trace, capturing the relationships between spans and logs within the trace. Third, Model Training leverages GNN and PU learning for trace-based anomaly detection, enabling the generation of vector representations for each trace based on its SCG. PU learning allows the model to train with a limited set of labeled anomalous traces, estimating empirical risk from the data available. Next, the trained model processes the SCG and utilizes non-parametric unsupervised learning to determine the threshold. MS-FLAD then provides a prediction regarding the presence of anomalies. In the following, we describe each component in more detail.



(a) Log parsing

(b) Workflow of semantic vectorization

Figure 4.4: Log Embedding

4.5.1 Log and Trace Embedding

In this section, we describe the log embedding process, which parses raw log messages. Next, semantic vectorization turns these messages into fixed-dimension vectors. Then, we focus on tracing embedding, which is essential for understanding system causal relationships and extracting vectors for each span.

4.5.1.1 Log Embedding

Given that raw log messages have an unstructured nature, automated log analysis encounters challenges. Unstructured log messages introduce difficulties in parsing and extracting meaningful information, attributed to the lack of standardization. The inherent variability, limited searchability, and reduced readability further compound these challenges, hindering the efficiency of log analysis processes. To address this, a log parsing strategy is employed to abstract the parameters within each log message. This transforms the data into a structured format, facilitating subsequent analysis. We adopt the so-called Drain [86], a technique known for its high parsing accuracy and efficiency in processing unstructured log data in real-time streams. To facilitate an integrated analysis with trace data, we first identify and capture the span ID and trace ID linked to each log message before initiating the parsing process. Following the parsing step, both the trace ID and span ID are appended to every extracted log event.

Fig. 4.4a presents a log data example from the TrainTicket application, simplifying some fields for clarity [85]. Log messages, shown on the console, include a consistent log event and a variable log parameter, the latter capturing attributes like contact ID and balance value. Log parsing, as executed by Drain, extracts log events from messages, facilitating their conversion into structured sequences, as depicted in Fig. 4.4a a log sequence consists of log events that track a specific task's execution flow. Each event in the sequence shares

the same task ID, allowing for their chronological association.

4.5.1.2 Semantic Vectorization

Semantic vectorization extracts semantic information from log events, transforming it into a fixed-dimension vector, known as a semantic vector. This transformation takes place independently of the log event's previous occurrences. The semantic vectorization workflow is depicted in Fig. 4.4b and encompasses three key steps: log events pre-processing, word vectorization, and TF-IDF-based aggregation, which are explained in more detail next.

(A) Log Events Pre-processing: To determine the semantics of log events, we treat each log event E as a natural language sentence, expressed as $S = [t_1, t_2, \dots t_N]$, where token i is represented by t_i, $\forall i \in [1, N]$ and N indicates the total length of the log event sentence. Although most tokens are valid English words, log events may contain non-character tokens and various variable names. The pre-processing for each log event sentence involves several steps. First, we eliminate all non-character tokens from sentences S, including delimiters, operators, punctuation, and numerical digits. Then, we discard common stop words such as “a" and “the". Finally, we split composite variable names in log events into their basic components, e.g., "TypeDeclaration" becomes "type" and "declaration", and "isCommitable" splits into "is" and "Commitable". This method deconstructs compound tokens into their elemental parts.

(B) Word Vectorization: After pre-processing, each log-event sentence S is converted into a semantic vector V. This transformation leverages pre-trained word vectors from FastText to capture the semantic meaning of log events, including semantic similarity among words. FastText assigns each word a D-dimensional vector, with $D = 300$ for its word vectors. Consequently, a log-event sentence S is transformed into a list of word vectors $\mathcal{U} = [\boldsymbol{u}_1, \boldsymbol{u}_2, \dots, \boldsymbol{u}_N]$, where each word is replaced by its corresponding vector. The word vector for t_i is represented as $\boldsymbol{u}_i \in \mathbb{R}^d, i \in [1, N]$, and N represents the total number of tokens in the log-event sentence.

(C) TF-IDF Aggregation: To ensure a fixed-dimensional representation for log events, which may vary in word count N, we aggregate the N word vectors in $\mathcal{U}$ into a D-dimensional vector. This method guarantees the uniform D for all semantic vectors, even with different N values. In log events, not all words carry the same level of significance. Commonly occurring words, such as “api" and “get", are less crucial compared to less frequent words like “food" and “submit" in the context of log embedding. Therefore, we apply weighted aggregation using the Term Frequency-Inverse Document Frequency (TF-IDF) [87]. This technique allows for precise weighting of words based on their significance,

where the TF for a word is defined as:

$$\text{TF(word)} = \frac{\text{Frequency of word in } S}{N}. \tag{4.4}$$

Moreover, to account for words like "block" that may appear in all log events and thus have reduced distinctiveness, we integrate the Inverse Document Frequency (IDF) metric, defined as IDF(word)= $\log\left(\frac{L'}{L'_{\text{word}}}\right)$, where L' is the total number of log events and L'_{word} is the number of log events containing the word. The TF-IDF weight for each word is then determined by TF(word) $\times$ IDF(word), where $\times$ represents the multiplication operation. Finally, by adding up the word vectors in $\mathcal{U}$ according to TF-IDF weights (w), the semantic vector $U \in \mathbb{R}^d$ that corresponds to a specific log event can be obtained as follows:

$$U = \sum_{i=1}^{N} w_i \cdot \boldsymbol{u}_i \tag{4.5}$$

where the dot operation represents scalar multiplication and N is the total number of words in the log event. This semantic vector effectively identifies log events with semantic similarities while distinguishing between different log events.

4.5.1.3 Trace Embedding

A trace consists of multiple spans that illustrate the causal relationships between different service calls. Each span corresponds to a specific service invocation and captures information such as the service and operation names, start time, duration, and status code. Span embedding refers to the process of capturing and encoding these invocation details to generate a trace graph representation. This involves encoding service and operation names, start and duration times, and status codes individually. These elements are then combined into a vector format for each span, providing a foundation for further analysis and interpretation of the trace data. Span embedding is carried out in three primary stages: (i) Semantic Embedding, (ii) Time Embedding, and (iii) Status Code Embedding, which are described in more detail next.

(A) Semantic Embedding: The semantic embedding component is responsible for producing a vector representation for each span in a trace by combining the service name and operation name. To start, we split these names into individual words using common delimiters found in microservices, such as “/”, “-”, and “:”. All words are then converted to lowercase, and non-alphabetical symbols, including punctuation marks and numbers, are removed. For example, the name “ts-travel-service/POST:/api/v1/travelservice/travelPlan/cheapest” is segmented into “ts”, “travel”, “service”, “post”, “api”, “v1”, “travelservice”, “travelplan”, and “cheapest”. To manage the dynamic and variable nature of service and operation names in microservices, we apply

the WordPiece tokenization algorithm [88], which divides words into smaller tokens based on character sequences. This technique helps handle out-of-vocabulary (OOV) words. For instance, "traveldate" and "trainnumber" are tokenized into "travel", "date", "train", and "number". Using the token sequence obtained from this process, we then utilize a pre-trained BERT model [89], a transformer-based language representation model, to generate vector embeddings. Specifically, we use the BERT-Base model [89], which includes 12 transformer encoder layers and has a hidden layer size of 768 dimensions. This model generates a 768-dimensional vector for each span, capturing the semantic details of both the service and operation names. Leveraging the BERT-Base model's 12-layer transformer architecture and its 768-dimensional hidden layer has proven effective for comprehending complex language patterns and enhancing performance in tasks requiring detailed semantic understanding.

(B) Time Embedding: Each span in the system records crucial details about a service invocation, such as the start time and duration of the interaction between the caller and callee. To improve the detection of time-related anomalies, we extract four key time-based attributes from these timestamps: (1) duration time, representing the total time of the span; (2) waiting time, which is the period the callee waits for a response from other services; (3) local execution time, referring to the time taken by the callee to complete the current invocation, excluding the waiting time; and (4) relative start time, which measures the time difference between the start of the span and the start of the relevant parent span.

With the BERT-Base model, each span is represented by a 768-dimensional vector that captures the embedding of both the service and operation names. However, if time features are limited to just one dimension within this 768-dimensional vector, there is a risk that the time-related characteristics could be underrepresented in the span's overall embedding. Moreover, the significant variation in time durations across different traces (ranging from just a few milliseconds to several thousand) complicates the process of weight convergence and reduces the model's training efficiency. To address these issues, we project a single-dimensional time feature $\boldsymbol{t}$ into a 768-dimensional vector space denoted by $\boldsymbol{E}_{\text{time}}$. Subsequently, we use the softmax function to create a soft one-hot encoding, where each element lies between 0 and 1 and the sum of all elements is equal to 1. The soft one-hot encoding vector $\boldsymbol{s}$ is obtained as follows:

$$\mathbf{s} = \tau(\boldsymbol{t}\boldsymbol{W} + \boldsymbol{b}), \tag{4.6}$$

where $\tau(\cdot)$ is the softmax function, $\boldsymbol{W} \in \mathbb{R}^p$ is the weight matrix, and $\boldsymbol{b} \in \mathbb{R}^p$ is the bias. Next, we project the vector $\mathbf{s}$ into a vector space specifically designed for time embeddings. The soft one-hot encoding $\mathbf{s}$ is multiplied by the time embedding vector $\boldsymbol{E}_s \in \mathbb{R}^{p \times d}$, which results in p-dimensional vector $\boldsymbol{E}_{\text{time}}$ as follows:

$$\boldsymbol{E}_{\text{time}} = \mathbf{s} \odot \boldsymbol{E}_s, \tag{4.7}$$

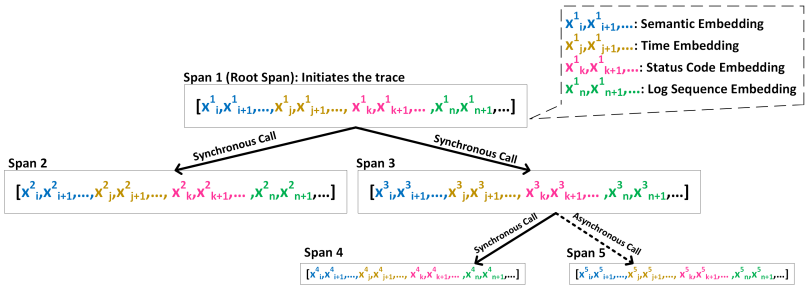


Figure 4.5: An Example of an SCG in TrainTicket application.

where $\odot$ denotes the element-wise multiplication of two vectors of the same length. Finally, to create a comprehensive representation of the time-related attributes within a span, we concatenate the four time embedding vectors, i.e., duration time, waiting time, local execution time, and relative start time (which are defined above). This combined representation effectively captures and encodes the temporal information associated with the span.

(C) Status Code Embedding: The HTTP/1.1 standard [90] defines a total of 63 status codes, organized into five main categories. We apply one-hot encoding to represent these status codes, where each status code is mapped to a 63-dimensional vector. In this representation, each dimension corresponds to a specific status code. For example, a status code of 200 is encoded as a vector with a value of 1 in the 5th dimension and 0 in all other dimensions, while a status code of 404 is encoded with a value of 1 in the 28th dimension and 0 in all other dimensions. This method of encoding status codes enables an efficient analysis of HTTP traffic.

4.5.2 Graph Building

Traces have a hierarchical structure consisting of service invocations, known as spans. In our proposed MS-FLAD method, this hierarchical structure is efficiently modeled using an SCG, a directed acyclic graph. In this graph, each node corresponds to a span within the trace, while the edges represent the parent-child relationships between spans. A directed edge from Span 1 to Span 2 signifies that Span 1 is the parent of Span 2. The vector representation of each span is assigned as an attribute to the corresponding graph node to capture its specific characteristics. Fig. 4.5 illustrates an example of an SCG, which corresponds to the trace shown in Fig. 4.3. In this figure, each rectangle represents a span, and each edge signifies a causal relationship between two spans. For instance, Span 1 represents the root span of the trace, and Spans 2 and 3 are child spans of Span 1. Each span's vector representation comprises four components, i.e., semantic, time, status code, and log sequence.

4.5.3 Model Training

In our proposed approach, trace-based anomaly detection is formulated as a PU learning problem, leveraging historical knowledge of anomalous traces while minimizing the number of labeled traces used for training. PU learning involves training a binary classifier with a small set of positive samples (i.e., anomalous traces) and a larger set of unlabeled samples. Traces are represented as SCGs, where span and log embeddings are used as node attributes. To derive meaningful trace representations, we utilize a Graph Neural Network (GNN) known as the Graph Attention Network (GAT), which incorporates the multi-head self-attention mechanism [91]. We selected GAT because it is well-suited to the graph structure of trace data, enabling it to learn vector representations of traces while capturing complex relationships and dependencies within the graph. These GNNs produce vector representations of the traces, as illustrated in Fig. 4.3. For training the binary classifier in trace-based anomaly detection, we adopt the non-negative risk estimator (nnPU) algorithm [92], which is known for its robustness against overfitting.

Let $g = V, A, X$ represent an SCG, where V is the collection of nodes, A is the adjacency matrix that defines the edges, and X is the set of attributes for each node, with h_i denoting the vector representation of each node. In an SCG, the GAT layer computes attention scores for neighboring nodes, highlighting their relative importance. We obtain the attention score e_{ij} from node j to node i as follows:

$$e_{ij} = \psi\left(\mathbf{a}^T \cdot (\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j)\right), \tag{4.8}$$

where $\psi(\cdot)$ is the LeakyRelu activation function, $\|$ represents concatenation, $\mathbf{h}_i$ and $\mathbf{h}_j$ denote the vector representations of node i and node j, respectively. The weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ corresponds to a shared linear transformation, where F is the dimensionality of the input node features and F' is the dimensionality of the transformed features. $\mathbf{a}$ is a learnable attention vector. To calculate the attention coefficients α_{ij} from node j to node i, a softmax function is used to normalize the importance across all neighboring nodes of i:

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{c \in \mathcal{N}_i} \exp\left(e_{ic}\right)}, \tag{4.9}$$

where $\mathcal{N}_i$ represents the neighborhood of node i. The GAT utilizes multi-head attention to enhance the stability of the attention mechanism's learning process. The attention coefficients are used to compute the output node representation h'_i as follows:

$$\mathbf{h}'_i = \|_{c=1}^{C} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^c \mathbf{W}^c \mathbf{h}_j\right), \tag{4.10}$$

where C denotes the number of attention heads involved. Each attention head, represented by α_{ij}^c, has its own attention score. The weight matrix $\mathbf{W}^c$ corresponds to the linear

transformation for attention head c. $\sigma(\cdot)$ refers to the activation function.

After executing the computations through m GAT layers, MS-FLAD generates vector representations for each node. The overall graph representation, v_g, is derived by averaging the vectors across all nodes as follows:

$$v_g = \frac{1}{N_g} \sum_{n=1}^{N_g} \mathbf{h}_n'^m, \tag{4.11}$$

where N_g denotes the number of nodes in graph g. The vector representation $\mathbf{h}_n'^m$ represents the output of node n from the embeddings of GAT layer m.

During the training phase, our proposed MS-FLAD utilizes non-negative risk estimation from the nnPU algorithm [92], a large-scale PU learning technique, to iteratively optimize the GAT parameters. In each epoch, the training dataset is split into N mini-batches, and MS-FLAD updates the GAT parameters based on the risk estimation for each mini-batch. Consider a two-layer perceptron (MLP) function f with a single output dimension, and let L represent the sigmoid loss function. Each mini-batch contains n_p labeled anomalous traces, where p denotes positive (anomalous) data, and u refers to the unlabeled data. The estimated risk $\widehat{R}_{\mathrm{p}}^{+}$ associated with labeled anomalous traces is obtained as follows:

$$\widehat{R}_{\mathrm{p}}^{+} = \frac{1}{n_p} \sum_{i=1}^{n_p} L\left(f\left(v_i^p\right), +1\right). \tag{4.12}$$

Similarly, the estimated risk $\widehat{R}_{\mathrm{p}}^{-}$ associated with misclassifying labeled positive (anomalous) traces as negative is obtained as follows:

$$\widehat{R}_{\mathrm{p}}^{-} = \frac{1}{n_p} \sum_{i=1}^{n_p} L\left(f\left(v_i^p\right), -1\right). \tag{4.13}$$

We calculate the estimated risk $\widehat{R}_{\mathrm{u}}^{-}$ for unlabeled traces by treating them as normal as follows:

$$\widehat{R}_{\mathrm{u}}^{-} = \frac{1}{n_u} \sum_{i=1}^{n_u} L\left(f\left(v_i^u\right), -1\right), \tag{4.14}$$

where n_u is the number of unlabeled traces. Finally, the empirical risk estimation $\widehat{R}_{\mathrm{pu}}$ is given by:

$$\widehat{R}_{\mathrm{pu}} = \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{+} + \widehat{R}_{\mathrm{u}}^{-} - \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-}, \tag{4.15}$$

where the hyperparameter π_{p} denotes the class prior probability of anomalous traces.

Let β be the hyperparameter which ensures the risk is non-negative. If $\widehat{R}_{\mathrm{u}}^{-} - \pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-} \geq -\beta$, MS-FLAD uses Adam [93] optimization to minimize $\widehat{R}_{\mathrm{pu}}$ and optimize GAT parameters; otherwise, Adam is used to minimize $\pi_{\mathrm{p}} \widehat{R}_{\mathrm{p}}^{-} - \widehat{R}_{\mathrm{u}}^{-}$ and optimize the GAT parameters.

4.5.4 Anomaly Detection

The anomaly detection component is responsible for generating the final outcome. To detect anomalies, we apply the sigmoid function to the output of the MLP. Let y_i be the output of the MLP for $f(v_i)$. The anomaly probability q_i is given by:

$$\mathrm{q}_i = \sigma(y_i) = \frac{1}{1 + e^{-y_i}}, \tag{4.16}$$

To detect anomalies, a threshold T_e is considered; if $q_i > T_e$, an anomaly is detected.

4.5.4.1 Initial threshold setting

We build a set of anomaly probabilities $\mathcal{Q} = \{q_1, q_2, \ldots, q_{|\mathcal{Q}|}\}$, where $q_i \in \mathcal{Q}$, $\forall i \in \{1, \ldots, |\mathcal{Q}|\}$ and we define K-means clustering function $\mathcal{F}(q_i)$ that maps each q_i to a cluster $C_x \in \mathcal{C}$, as follows:

$$\mathcal{F}(q_i) = C_x,\ \forall i \in \{1, \ldots, |\mathcal{Q}|\},\ \forall x \in \{1, \ldots, |\mathcal{C}|\}. \tag{4.17}$$

Depending on the value of q_i, the system can fall in one of the following states: (i) normal, (ii) transitive, or (iii) anomalous. We set the initial cluster centers to 0, the average of all q_i in $\mathcal{Q}$, and the maximum q_i value in $\mathcal{Q}$ for normal, transitive, and anomalous states, respectively. Let $\mathcal{Q}_a$ be the set obtained for the anomalous cluster. The initial threshold T_e^0 is then obtained as follows:

$$T_e^0 = \min(\mathcal{Q}_a). \tag{4.18}$$

4.5.4.2 Threshold calculation

To refine the anomaly detection, we generate a series of candidate thresholds T_z as:

$$T_z = T_e^0 + z \times d,\ \forall z \in \{0, \ldots, z_{\max}\} \tag{4.19}$$

where z is the smallest integer such that $T_z > max(Q)$, and d is the step size. Large values of d may increase false positives, while smaller values of d may increase the execution time of the algorithm. For each candidate threshold T_z, the set $\mathcal{Q}$ is divided into two subsets, $\mathcal{Q}_N = \{q_i \in \mathcal{Q} \mid q_i \leq T_z\}$ and $\mathcal{Q}_A = \{q_i \in \mathcal{Q} \mid q_i > T_z\}$. The updated T_z' is then calculated for each T_z as follows:

$$T_z' = \frac{\sigma(\mathcal{Q})}{\sigma(\mathcal{Q}_N)} \times \frac{1}{|\mathcal{Q}_A|}, \tag{4.20}$$

where $\sigma(\cdot)$ represents the standard deviation. The optimal threshold T_e is determined by selecting the candidate T_z that maximizes T_z' as follows:

$$T_e = \arg\max_{T_z}(T_z'). \tag{4.21}$$

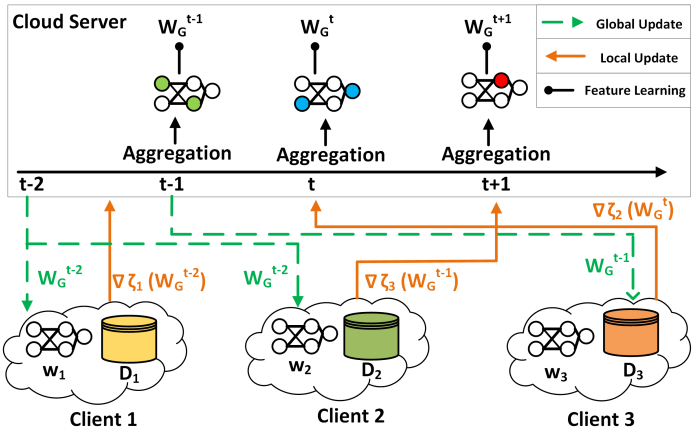


Figure 4.6: ART-FL overview.

4.5.5 Global Model Aggregation and Update

Up until now, we have explained the entire process of the anomaly detection model at the edge client. This model is then extended for use in an FL framework. We propose an asynchronous real-time FL, where the server updates the global model $\mathbf{w}_G$ as soon as it receives updates from any client, without waiting for the others. Each client maintains a local copy of $\mathbf{w}_G$, which may differ from those of other clients due to asynchronous updates. The server's task includes not only aggregating these updates but also performing feature learning to extract and enhance cross-client feature representations, thereby improving the overall model performance. Fig. 4.6 shows the update process of the ART-FL method. The server aggregates the updates from the clients as they arrive and conducts feature learning on the aggregated parameters to create a cross-client feature representation. The updated global model $\mathbf{w}_G$ is then sent to clients that are prepared for the next iteration. As clients collect new data samples, we implement a decay coefficient to balance the previous and current local gradients through a repetitive local computation process. Algorithm 4.1 depicts the pseudocode of the proposed ART-FL method. Learning on the cloud server is shown in lines 2-7 of the Algorithm 4.1, which involves aggregating and processing data received from the local clients. Learning on local clients is described in lines 8-15 of the Algorithm 4.1, where the focus is on updating local models based on individual client data and the aggregated information received from the cloud server. As shown in Fig. 4.6, when the server receives updates from slower clients (e.g., Client 2), it may have already performed multiple updates to the global model $\mathbf{w}_G$. This asynchronous updating issue, which occurs due to data heterogeneity and network delays, is typical in real-world scenarios. To cope with this, we develop a global feature representation on the server and implement a dynamic step size to train local clients.

Algorithm 4.1: ART-FL

Require: multiplier r_k, learning rate $\bar{\eta}$, regularization parameter λ , decay coefficient Γ.
Ensure: Updated global model $\mathbf{w}_G$
1: Initialize: $h_k^p = h_k = 0$, $v_k = 0$, $r_k^t = \max\left\{1, \log\left(\bar{d}_k^t\right)\right\}$ where $\bar{d}_k^t = \frac{1}{t}\sum_{\tau=1}^{t} d_k^\tau$.
2: [**Procedure at Cloud Server**]
3: **for** $t = 1, 2, \ldots, T$ (global iterations) **do**
4: compute $\mathbf{w}_G^t$ */* get the update on $\mathbf{w}_G^t$ */* ▷ [Eq.(4.22)]
5: apply feature learning to updated $\mathbf{w}_G^t$ ▷ [Eq.(4.23)]
6: **end for**
7: [**Procedure of Local Client** k]
8: receive $\mathbf{w}_G^t$ from the server
9: Compute $\nabla s_k = \nabla f_k(\mathbf{w}_k^t) + \lambda(\mathbf{w}_k^t - \mathbf{w}_G^t)$
10: Set $h_k^p = h_k$
11: Update gradient $\nabla\zeta_k \leftarrow \nabla s_k - \nabla s_k^p + h_k^p$
12: Update local model $\mathbf{w}_k^{t+1} \leftarrow \mathbf{w}_k^t - r_k^t\bar{\eta}\nabla\zeta_k$
13: Update $h_k = \Gamma h_k + (1-\Gamma)v_k$
14: Compute $v_k = \nabla s_k(\mathbf{w}_G^t; \mathbf{w}_k^t)$
15: upload $\mathbf{w}_k^{t+1}$ to the server

4.5.5.1 Learning on the Cloud Server

Each update cycle on the server begins by aggregating updates from client k as follows:

$$\begin{aligned}\mathbf{w}_G^{t+1} &= \mathbf{w}_G^t - \frac{n_k'}{N'}\left(\mathbf{w}_k^t - \mathbf{w}_k^{t+1}\right)\\ &= \mathbf{w}_G^t - \frac{n_k'}{N'}\left(\mathbf{w}_k^t - \left(\mathbf{w}_k^t - \eta_k^t\nabla\zeta_k\left(\mathbf{w}_G^t\right)\right)\right)\\ &= \mathbf{w}_G^t - \eta_k^t\frac{n_k'}{N'}\nabla\zeta_k\left(\mathbf{w}_G^t\right),\end{aligned} \tag{4.22}$$

where η_k^t is the learning rate of client k at iteration t, n_k' represents the number of new data received from client k, N' is the updated total number of samples across all clients, and $\nabla\zeta_k$ denotes the gradient of the loss function for client k. Following this aggregation, the server applies feature representation learning techniques to address any potential performance issues arising from asynchronous updates. We perform feature extraction on the first layer (e.g., GAT) to generate the feature representation and denote the parameters of this layer as $\mathbf{w}_{(1)}^{t+1}$. For each element $\mathbf{w}_{(1)}^{t+1}[i,j]$ in column $\mathbf{w}_{(1)}^t[j]$ of $\mathbf{w}_{(1)}^t$, the updated $\mathbf{w}_{(1)}^{t+1}$ is obtained as follows:

$$\mathbf{w}_{(1)}^{t+1}[i,j] = \alpha_{(1)}^{t+1}[i,j] \times \mathbf{w}_{(1)}^t[i,j], \tag{4.23}$$

where $\alpha_{(1)}^{t+1}[i,j]$ is computed through a softmax function over the absolute values of the weights as follows:

$$\alpha_{(1)}^{t+1}[i,j] \leftarrow \frac{\exp\left(\left|\mathbf{w}_{(1)}^t[i,j]\right|\right)}{\sum_j \exp\left(\left|\mathbf{w}_{(1)}^t[i,j]\right|\right)}. \tag{4.24}$$

4.5.5.2 Learning on Local Clients

Local clients aim to align their models closely with the global model by addressing local deviations and incorporating regularization. The local objective function s_k, which penalizes large deviations between local and global models, is given by:

$$s_k(\mathbf{w}_k) = f_k(\mathbf{w}_k) + \frac{\lambda}{2}\|\mathbf{w}_k - \mathbf{w}_G\|^2, \tag{4.25}$$

where $f_k(\mathbf{w}_k)$ is the local loss function for client k and λ is the regularization parameter.

Given that the data continues arriving at local clients during the training process, each client is required to perform real-time learning. Each client retrieves the latest global model $\mathbf{w}_G^t$ from the cloud server and updates it with new local data. This requires balancing the impact of the previously learned parameters with the newly updated data using a decay coefficient Γ. At each global iteration t, client k receives the global model $\mathbf{w}_G^t$ from the server. Let ∇s_k^{p} denote the gradients computed during the previous update at client k. The optimization process at client k during this iteration involves adjusting the local model based on the new gradients and a decay mechanism to blend historical gradient influences with current data as follows:

$$\nabla \zeta_k \leftarrow \nabla s_k - \nabla s_k^{\mathrm{p}} + h_k^{\mathrm{p}} \tag{4.26}$$

where ∇s_k is the gradient of the local objective function for client k and h_k^{p} is initialized to zero and updated each iteration to balance the old and new gradients using the decay coefficient Γ as follow:

$$h_k^{\mathrm{p}} = \Gamma h_k^{\mathrm{p}} + (1-\Gamma)\nabla s_k^{\mathrm{p}} \tag{4.27}$$

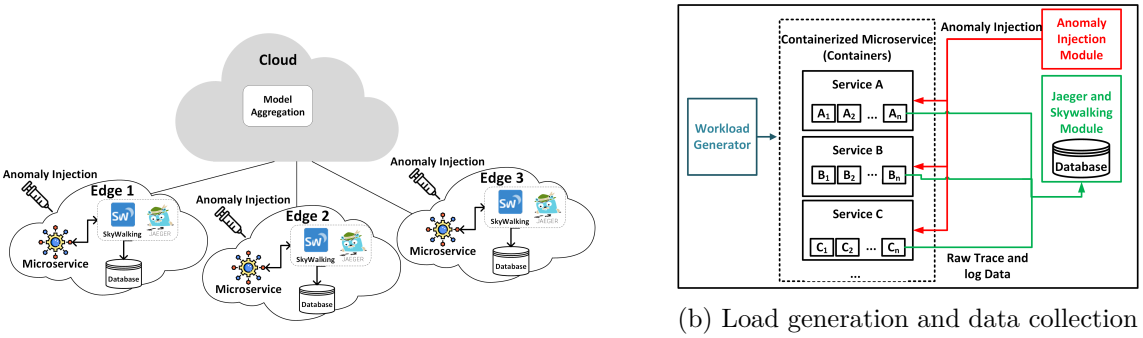
The learning rate η_k^t is employed to apply these adjustments to the local model. The local model $\mathbf{w}_k^{t+1}$ is then updated as follows:

$$\begin{aligned} \mathbf{w}_k^{t+1} &= \mathbf{w}_k^t - \eta_k^t \nabla \zeta_k(\mathbf{w}^t), \\ &= \mathbf{w}_k^t - \eta_k^t(\nabla f_k(\mathbf{w}_k^t) - \nabla s_k^{\mathrm{p}} + h_k^{\mathrm{p}} + \lambda(\mathbf{w}_k^t - \mathbf{w}^t)), \end{aligned} \tag{4.28}$$

where λ is a regularization parameter that helps mitigate overfitting by penalizing large deviations from the global model.

Variability in client update frequencies due to factors such as bandwidth, network delays, and data heterogeneity can affect model training. To cope with this, we implement a dynamic learning step size. If a client has a larger data volume or faces communication issues, its activation rate will be low, so we increase its learning step size accordingly. This dynamic adjustment is represented by r_k^t, which is calculated to reflect the activation frequency and network conditions of client k at iteration t. Initially, all clients start with a baseline learning rate $\eta_k^t = \bar{\eta}$. Thus, Eq. (4.28) for client k's model at iteration t is updated as follows:

$$\mathbf{w}_k^{t+1} = \mathbf{w}_k^t - r_k^t \bar{\eta} \nabla \zeta_k(\mathbf{w}_G^t), \tag{4.29}$$



(a) Lab setup for evaluation

(b) Load generation and data collection from the testbed

Figure 4.7: Experimental testbed for distributed traces in ML-based anomaly detection.

where $r_k^t = \max\{1, \log(\bar{d}_k^t)\}$ and $\bar{d}_k^t = \frac{1}{t}\sum_{\tau=1}^{t} d_k^\tau$ is the average delay over the past t iterations. $\nabla\zeta_k$ includes contributions from the gradient of the loss function and any regularization terms, adjusted dynamically by r_k^t. This dynamic adjustment helps mitigate the impact of asynchronous updates based on past communication delays and varying data distributions across clients, ensuring more stable and efficient convergence.

4.6 Performance Evaluation

In this section, we describe the implementation details, present the findings, and conclude with a discussion.

4.6.1 Experiment Settings

We present a summary of our experimental setup, detailing the benchmark application used, the load generation and trace collection methods, the specifications for anomaly injection, and the parameter configurations and coding environment employed to implement the proposed solution.

4.6.1.1 Benchmark Application

In our evaluations, we utilized TrainTicket [85], a dynamic real-world benchmark designed specifically for microservices. TrainTicket simulates the core functions of a train ticket booking system, including ticket searches, reservations, payments, modifications, and user notifications. It adheres to microservice architecture principles and supports different interaction modes, such as synchronous and asynchronous calls, as well as message queues. The system consists of 41 business logic microservices (excluding database and infrastructure services) and is specifically developed to enable testing and experimentation with microservices and cloud-native technologies.

4.6.1.2 Experimental Testbed

Fig. 4.7a depicts our lab setup, which consists of a lab testbed environment established within Ericsson Research's private cloud, also known as Xerces. Xerces is powered by an infrastructure that includes approximately 300 servers, all orchestrated by an Infrastructure-as-a-Service (IaaS) platform based on OpenStack. Our test setup consists of a Kubernetes cluster comprising four sub-clusters with a total of 10 VMs operating on Ubuntu 20.04. The central cluster houses one VM designated as the Kubernetes master node. In each of the three edge clusters, two VMs are serving as Kubernetes worker nodes, along with a master node. These virtual machines' specifications are listed in Table 4.2. For monitoring and trace gathering within these Kubernetes clusters, we utilized Jaeger[2]. Additionally, Apache SkyWalking[3] is employed as our distributed tracing platform for log collection. Each cluster is equipped with one instance of both Jaeger and SkyWalking. The TrainTicket application is deployed on all master nodes. Kubernetes Cluster Federation (Kubefed) is used to streamline application deployment processes, such as setting up Jaeger and SkyWalking services across various clusters. Kubefed offers a centralized approach to manage and configure multiple Kubernetes clusters through a single API set, operating from the central site in our configuration.

Table 4.2: Configuration parameters and settings for the lab setup.

Site	Parameter	Value
Cloud Server	Number of VMs	1 VM
	CPU	4 cores (core i7-8700)
	RAM	16 GB
	HDD	40G
Edge	Number of VMs	9 VMs
	CPU	4 cores for each VM
	RAM	8 GB for each worker VM 16 GB for master VM
	HDD	40 GB for each VM

4.6.1.3 Load Generation and Data Collection

Fig. 4.7b illustrates our approach for generating workloads and collecting data from Kubernetes clusters. As explained in Section 4.6.1.2, we deploy a container-based application on three Kubernetes clusters. In Fig. 4.7b, a container is defined as a grouping of one or multiple containers forming a complete microservice, aligning with the Kubernetes concept of a pod. To enhance the diversity of collected data, we incorporate an anomaly injection module, injecting various anomalies into the containers. Each Kubernetes cluster comprises multiple nodes (hosts), with each node accommodating several containers. In our

[2] https://www.jaegertracing.io/
[3] https://skywalking.apache.org/

setup, each container hosts a specific microservice, and a microservice can be concurrently deployed across multiple containers. We employ a distributed tracing module, utilizing Jaeger and SkyWalking, on every VM to trace request flows within microservices. A workload generator is deployed to emulate real user requests. We simulate a range of user behaviors to generate realistic workloads, from basic tasks like visiting the homepage and searching for train schedules to more complex actions such as user authentication and purchasing tickets. Furthermore, we implement a dynamic scaling mechanism for the simulated user base, tailoring the frequency of each user behavior over time. This methodology facilitates the creation of a dynamic, varied mix of requests, effectively simulating real-life web traffic. For the generation of these synthetic workloads, we deploy the Performance Testing-based Application Monitoring (PPTAM) framework [94], which delineates five unique user profiles. Minor adjustments were made to PPTAM to allow for real-time variation in the user distribution across different types of requests. The generation of workload was evenly distributed across various request categories to ensure comprehensive coverage of all microservice benchmarks, as illustrated in Fig. 4.7b. From this setup, we successfully gathered 189,486 execution traces within the microservice architecture.

Table 4.3: List of injected anomalies into TrainTicket.

Object	**Anomaly Injection Operation**
Physical	CPU Overload: Inject a sudden spike in CPU usage in a VM. Memory Leak: Gradually consume more memory in a VM to mimic a memory leak. Network Congestion: Introduce network congestion by limiting bandwidth.
Database	Slow Query Processing: Drop indexes from some tables. Deadlock Problem: Connections lock tables, causing deadlocks. Data Flushing Fault: Truncate a table, slowing down insertions due to a storage engine bug.
Application	Unauthorized Access: Incorrect password for database access. Access Configuration Error: Incorrect user access authentication. Service Access Failure: Incorrect path set for microservices. Service Invocation Failure: Error in service implementation leading to inaccurate responses.
Pod	Pause Container: Use Docker's "PAUSE" command on a container. Error Hang: Trigger an error hang in a microservice container. CPU Load Increase: In a container, increase CPU load to 100%. Disable Pod: Randomly disable a specific pod.

4.6.1.4 Anomaly Injection

We implemented an anomaly injector with configurable options for selecting injection targets, anomaly types, injection timing, duration, and intensity. This injector is designed to be packaged as a file-system layer within microservice containers, enabling remote activation during the training phase. It includes different types of anomalies that have the potential to violate the SLOs, as shown in Table 4.3. Every service instance receives uniform random injections of anomalies of various kinds with tunable injection timing and intensity. We first chose a physical resource, database, application, and pod randomly and then applied the corresponding injection strategy randomly to the microservice. Every anomaly type has an

injection intensity randomly chosen from the range [0, 1]. Each anomaly injection in our experimentation has a duration of 3 minutes. To minimize the potential cross-influence of different injection operations, we ensure that the interval between successive injections is greater than 6 minutes, while the anomaly type and intensity are chosen randomly. 28.7% of the entire trace data consists of anomalous traces.

4.6.1.5 Parameter Setting and Coding Environment

The proposed MS-FLAD method was developed with PyTorch 1.10 and Python 3.10.9, employing PyTorch Geometric 2.2 for the GAT and adapting the nnPU algorithm [92] for PU learning. We incorporated semantic analysis using the pre-trained BERT-Base model [89] and WordPiece tokenizer [88], with time features embedded into 100 dimensions. The MS-FLAD method features a three-layer GAT with the first two layers using three attention heads and the final layer using one. Batch normalization followed each layer, and training parameters included a batch size of 128, a prior probability π_{p} of positive data of 0.15, and hyperparameter β set to 0. The training ran for 50 epochs with an Adam [93] optimizer learning rate of 0.001. Data was divided into training, validation, and testing sets in a 3:1:6 ratio, with $\sim$ 10% of anomalous traces in the training set labeled as positive, making up about 2% of the training data. To evaluate the detection performance, we also conduct training using a centralized variant of our proposed approach called Microservice-Anomaly Detection (MS-AD), where all the data is processed centrally without the constraints of data distribution and privacy concerns. As for the centralized MS-AD implementation, we utilize the complete original dataset for training on the cloud server in our testbed.

4.6.1.6 Evaluation Metrics

We consider precision, recall, and F_1-score to evaluate the performance of different anomaly detection methods under study. A large value of precision, recall, and F_1-score indicates that the model is efficient at flagging true anomalies while minimizing false alarms and missed anomalies, which is key for anomaly detection to meet the accuracy requirement. Considering anomalies as the positive condition, these metrics provide a comprehensive assessment of each method's ability to correctly detect True Positives (TP) while minimizing False Positives (FP) and False Negatives (FN).

Precision measures the proportion of correctly identified anomalous data sequences to all flagged anomalies (see Eq. 4.30), whereas recall captures the proportion of actual anomalous data that were correctly detected (see Eq. 4.31). The F_1-score represents the harmonic mean of precision and recall (see Eq. 4.32), providing a balanced score of the accuracy performance. These three metrics range from 0 to 1, where higher values indicate better performance. Note that precision focuses on reducing FP and recall on minimizing FN.

Table 4.4: Performance comparison for anomaly detection.

Methods	Effectiveness(%)			Time Efficiency	
	Recall	Precision	F_1-score	Training time (m)	Detection time (ms)
Microscope [52]	39.6	94.3	55.7	450	4.7
TraceAnomaly [50]	65.9	58.0	61.5	30	137
Multimodal Trace [51]	60.1	96.0	73.8	30.4	148
TDAD [3]	87.1	93.0	89.8	35	0.1
Seer [47]	67.8	93.8	78.7	32	98
FedAnomaly [64]	99.9	86.4	92.6	890	12
FL-SmartGrid [66]	96.2	87.5	91.7	82.1	2.2
MS-AD	95.0	87.1	90.9	67.1	0.7
Proposed MS-FLAD	98.7	94.0	96.3	21	0.4

$$Precision = \frac{TP}{TP + FP} \tag{4.30}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.31}$$

$$F_1\text{-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.32}$$

Moreover, time efficiency is measured using detection time, which is the time required to process a single trace of data, including the time needed for anomaly detection. A short detection time is crucial to meet the real-time and time-efficiency requirements of the anomaly detection system. Training time refers to the total time taken to train the model across all iterations until convergence. Moreover, scalability is evaluated in terms of how the response time of our anomaly detection model changes as trace size (i.e., number of events) increases. Finally, resource overhead is an important metric for ensuring cost-efficient communication and computation between edge clients and the cloud server. Communication overhead measures the amount of data exchanged between clients and the cloud server, while computation overhead refers to the processing time consumed by the clients to locally train the model before synchronization.

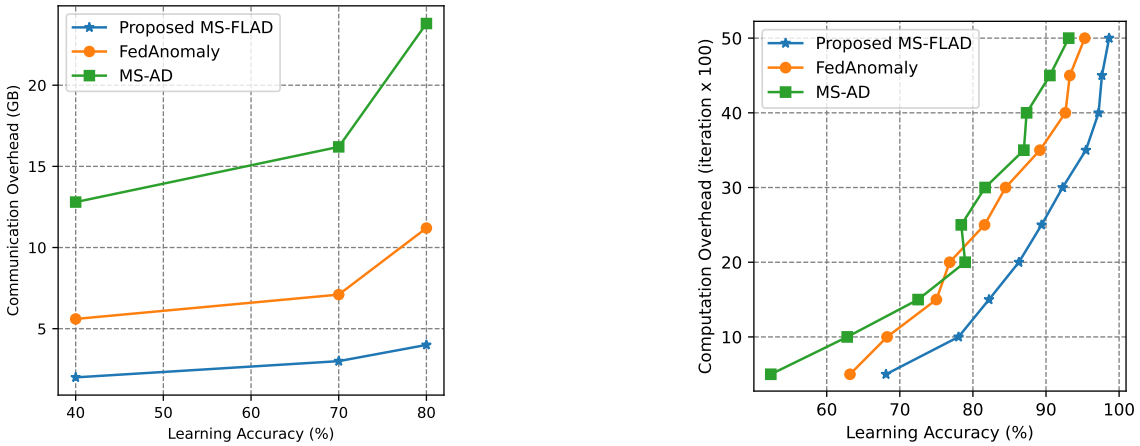
4.6.2 Evaluation Results

In this section, we present our evaluation results, which are categorized into two main sections. The first section presents the performance of anomaly detection in terms of effectiveness, time efficiency, communication overhead, and scalability. The second section examines the impact of parameter configuration on performance, along with an ablation study.

4.6.2.1 Anomaly Detection Results

(A) Effectiveness: We compare our proposed MS-FLAD anomaly detection algorithm with existing works, including Microscope [52], TraceAnomaly [50], Multimodal-Trace [51], TDAD [3], Seer [47], FedAnomaly [64], and FL-SmartGrid-AD [66]. Among these methods, FedAnomaly [64] and FL-SmartGrid-AD [66] are FL-based models, whereas the others are centralized methods, which are compared to the centralized variant MS-AD of our proposed method. Microscope [52], TraceAnomaly [50], MultimodalTrace [51], TDAD [3] rely on tracing data, LogAnomaly [70] only relies on log data and FedAnomaly [64], and FL-SmartGrid-AD [66] rely on metric data for anomaly detection. Table 4.4 illustrates the obtained results of recall, precision, and F_1-score for different anomaly detection solutions. The proposed MS-FLAD achieves a recall of 98.7%, precision of 94%, and an F_1-score of 96.3%. It is evident that our proposed MS-FLAD and MS-AD outperform other methods that depend solely on a single data source and utilize a centralized approach. This is because incorporating log messages and trace data in MS-FLAD, representing intra-service and inter-service behaviors, respectively, enables more timely and accurate detection of anomalies. Moreover, the proposed MS-FLAD and MS-AD outperform Seer [47], even though they utilized multiple source data (trace and metric). While Seer [47] focuses on detecting response time and invocation path anomalies in a unified manner, it does not consider the graph-based structure of traces and anomalies in resource usage. Our proposed MS-FLAD outperforms in precision, recall, and the F_1-score when compared to TDAD [3], Microscope [52], TraceAnomaly [50], and MultimodalTrace [51]. These methods are based on sequence-based trace representation, which is not suitable for capturing the causal connections among spans. The TraceAnomaly [50] only takes into account the response time and service sequences seen in traces. MultimodalTrace [51] only uses the operation sequences of traces to train an LSTM-based model. Additionally, these trace-based anomaly detection methods do not account for intra-service behavior. In contrast, our proposed MS-FLAD method considers the intra-service behavior of service instances. Therefore, MS-FLAD outperforms FedAnomaly [64] and FL-SmartGrid-AD [66] by achieving up to a 96.3% F_1-score. This is also because MS-FLAD incorporates GNN and PU learning to inherit the anomaly detection benefits of SCG.

(B) Time-Efficiency: Table 4.4 illustrates the training time and detection time for various methods under consideration. We observe from the table that our proposed MS-FLAD method, in comparison to trace-based methods such as TraceAnomaly [50], MultimodalTrace [51], shows a slower pace in model training, though it outperforms other methods in terms of detection time. Also, MS-FLAD is notably faster than all compared approaches in terms of detection time. We also observe from the table that the centralized variant MS-AD of our proposed method achieves a higher training time compared to



(a) (b)

Figure 4.8: Resource overhead analysis: (a) Communication overhead vs. learning accuracy and (b) learning accuracy vs. computation overhead.

other trace-based methods, but a lower detection time. This happens because the MS-AD method considers logs as well as traces, which incurs additional training time. This efficiency is attributed to the anomaly detection model being fully characterized by its network parameters θ, eliminating the need to store data for anomaly detection. Given that MS-FLAD is an FL approach, it is faster in training than the centralized approaches. Also, since the proposed MS-FLAD allows models to be trained locally, detection can occur in real time at the edge, where data is generated. This, as a result, minimizes the time it takes to detect anomalies. LogAnomaly [70], which is a log-based method, splits the event sequences into several subsequences for testing and training by using sliding windows. Each subsequence in this method utilizes count vector sequences, the dimensions of which are based on the number of log events. With our dataset containing over 800 log and span events, this approach encounters the curse of dimensionality. This issue is common in microservice systems with a large number of log or span events, rendering traditional log anomaly detection methods less effective in such environments. FL-SmartGrid-AD [66] is slower in training and testing than our proposed MS-FLAD while using a simpler sequence representation for traces and logs. This is due to the fact that GNNs treat each event in the graph as a network unit, allowing for concurrent message passing, while GRUs in FL-SmartGrid-AD [66] process each event in a long sequence serially, resulting in slower performance. Given that the FL approach shares computational resources with clients, the total time is notably reduced, even when factoring in the additional time required for FL averaging. Multiple computing instances from various clients sharing computational resources and parallel computing contribute to a reduction in the overall training time required to achieve optimal performance.

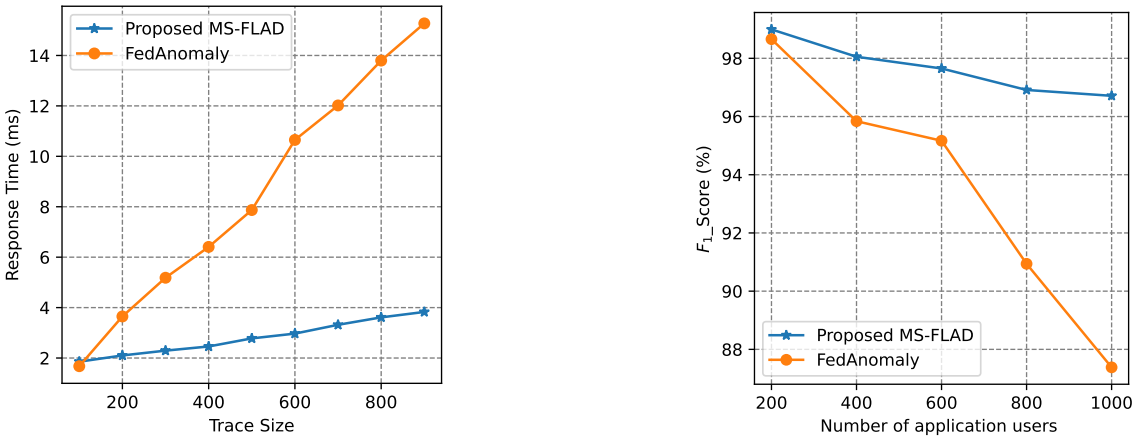
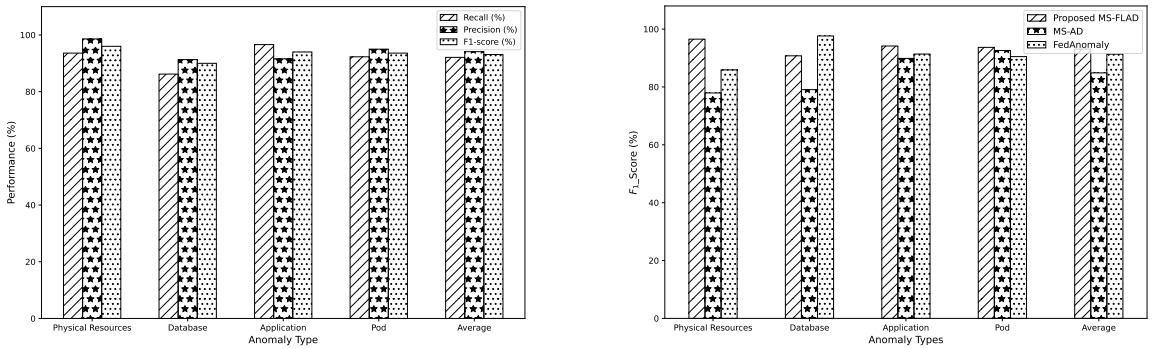


Figure 4.9: Scalability of the proposed MS-FLAD and FedAnomaly: (a) Response time of prediction vs. trace size, (b) F_1-score vs. the number of application users.

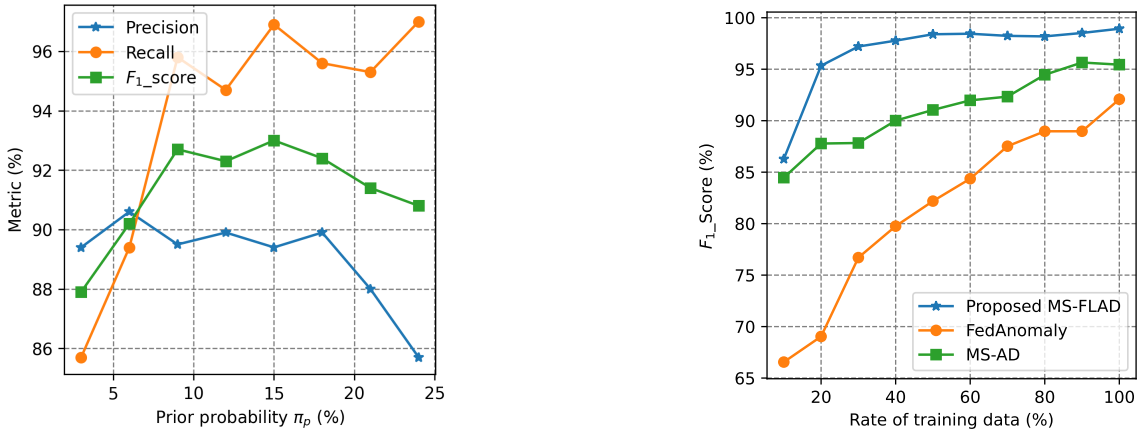
(C) Resource Overhead: Fig. 4.8a compares the communication overhead in gigabytes (GB) between MS-AD, FedAnomaly [64], and the proposed MS-FLAD at three distinct learning accuracy levels. MS-FLAD consistently maintains the lowest communication overhead, reducing it by approximately 3x to 4x compared to MS-AD, demonstrating its superior efficiency in minimizing communication costs while achieving high accuracy. Fig. 4.8b illustrates the comparison of the proposed MS-FLAD algorithm, FedAnomaly [64], and MS-AD in terms of learning accuracy as a function of computation overhead (iteration). The proposed MS-FLAD achieves the highest accuracy with faster convergence compared to the other methods.

(D) Scalability: Next, we examine how response time is affected by trace size, which is the total number of span and log events in the trace. Fig. 4.9a depicts trace size vs. response time. In our evaluations, we divided the range of event numbers from 0 to 900 into nine equal-size distinct groups, each spanning 100 events, and then randomly selected 100 traces from each group. For each trace, the anomaly detection model was run 300 times to determine the average response time for making a prediction. The evaluation focuses on the comparison of average response times between MS-FLAD and FedAnomaly [64] across traces of different sizes. This comparison was chosen due to the unique performance characteristics of each method, as previously detailed. Specifically, FedAnomaly [64] outperforms in detection accuracy, achieving a higher F_1-score than competing methods. Fig. 4.9a shows a linear increase in response time with trace size for both methods. Significantly, MS-FLAD



(a) (b)

Figure 4.10: Performance evaluation of MS-FLAD: (a) Recall, precision, and F_1-score across different anomaly types, and (b) F_1-score comparison across methods.



(a) (b)

Figure 4.11: Performance evaluation of MS-FLAD: (a) Impact of prior probability, π_p, on the performance, and (b) F_1-score vs. different training data rates.

maintains quicker response times in comparison to FedAnomaly [64].

Moreover, we evaluate the proposed MS-FLAD scalability under varying numbers of application users. We systematically increased the number of users from 200 to 1000 to simulate diverse scenarios of user engagement. As shown in Fig. 4.9b, the proposed MS-FLAD consistently maintained the F_1-score above 95%, even with the growing number of application users. This robust performance illustrates the method's reliability and scalability. When compared to the FedAnomaly [64] approach, our proposed MS-FLAD demonstrates superior performance across all user group sizes. Particularly, the gap in F_1-score between the two methods becomes more noticeable as the number of users increases.

4.6.2.2 Impact of Configuration Parameters on the Performance

The experimental results outlined in Fig 4.10a illustrate the effectiveness of our approach in detecting anomalies with high precision, recall, and F_1-score across various anomaly types. The results are categorized into four groups, namely anomalies related to physical resources, databases, applications, and pods. Furthermore, we conduct a comparative analysis against MS-AD and FedAnomaly [64], utilizing the F_1-score as a comprehensive metric capturing both precision and recall as shown in Fig. 4.10b. The average F_1-score across all anomaly types is presented. Our proposed MS-FLAD outperforms FedAnomaly [64] by 2% and outperforms MS-AD by 7%.

Fig. 4.11a illustrates performance vs. the prior probability π_p of anomalous traces as described in Eq. (4.15). The prior probability π_p represents an estimation of the proportion of anomalous traces within the dataset, which directly affects the effectiveness of the nnPU risk estimator during training. As shown in Fig. 4.11a, we can observe the influence of π_p on the performance of MS-FLAD, including precision, recall, and F_1-score. Generally, as π_p increases, recall tends to rise while precision decreases. This behavior is attributed to the model's inclination to predict more traces as anomalous with higher values of π_p. When π_p is approximately 15%, which closely matches the actual proportion of anomalous traces in the dataset (15.4%), MS-FLAD achieves its highest F_1-score. Furthermore, MS-FLAD consistently maintains high performance, with F_1-scores close to or exceeding 0.9, when π_p falls within the range of 2.5% to 25%. Consequently, MS-FLAD demonstrates strong performance when π_p falls within a reasonable range, with its optimal performance occurring when π_p aligns closely with the actual proportion of anomalous traces in the dataset.

To evaluate the incremental real-time learning process, we examined how the performance changes as the amount of training data grows. This is illustrated in Fig. 4.11b, where we conducted experiments using different proportions of the clients' training data. The results show that ART-FL consistently performs the best as more data becomes available. In contrast, FedAnomaly [64] shows significant fluctuations in performance due to the instability of synchronous approaches when local data increases. Compared to the MS-AD model, our asynchronous ART-FL approach maintains more stable performance as data grows.

4.6.2.3 Ablation Study

We perform an ablation study to assess the effects of different FL approaches on the model. We compare the proposed MS-FLAD with both synchronous and asynchronous FL approaches. The synchronous model, As shown in Table 4.5, the synchronous model, FedAvg, aggregates updates using a simple averaging method [95], resulting in a training time of 45. Despite this slower training, it achieves a recall of 55%, precision of 87%, and an F_1-score of 66%, indicating its challenges in efficiently managing distributed data. Conversely, the asynchronous model, FedProx [96], employs a weighted average

Table 4.5: Ablation performance of MS-FLAD with different FL approaches.

MS-FLAD Variant	**Recall (%)**	**Precision (%)**	**F_1-score (%)**	**Training Time (m)**
with FedAvg [95]	55	87	66	45
with FedProx [96]	65	84	72	26
with proposed ART-FL	98.7	94	96.3	21

to accommodate updates as they arrive, leading to a training time of 26. This model demonstrates better performance with a recall of 65%, precision of 84%, and an F_1-score of 72%, but still underperforms our ART-FL model. Lastly, our ART-FL model demonstrates a processing time of 21. It significantly enhances both speed and accuracy with a recall of 98.7%, precision of 94%, and an F_1-score of 96.3%, making it a competitive alternative that nearly rivals the performance of the global model.

4.7 Conclusion

In this chapter, we proposed an asynchronous FL method called ART-FL for detecting anomalies in trace and log data of microservice cloud applications. ART-FL enhances computational efficiency by allowing the cloud server to aggregate updates without waiting for slower clients. Our proposed method for edge clients combines GNN and PU learning for the accurate detection of anomalous traces. The training process in our proposed method involves the use of a small number of labeled anomalous traces as well as a relatively large number of unlabeled traces. Our trace-driven evaluations on a microservice benchmark demonstrate that the proposed method outperforms the existing anomaly detection methods by 4% in terms of F_1-score and by 5x in terms of detection time. Compared to other asynchronous methods, ART-FL delivers superior performance, particularly in managing local streaming data. Time efficiency assessments indicate that ART-FL operates faster than traditional synchronized FL approaches.

Chapter 5

Contextual Logs in ML-based Anomaly Detection[1]

5.1 Introduction

Log anomaly detection plays a crucial role in maintaining system reliability, identifying security threats, and enabling failure prediction. This is particularly vital in real-time cloud, where services operate under strict latency constraints and must process massive volumes of log data efficiently [71, 97, 98]. Effective anomaly detection in logs requires handling high-velocity data streams, adapting to dynamic system behavior, and reducing false positives, all while ensuring timely detection and mitigation. Despite its significance, log-based anomaly detection in real-time cloud faces challenges. First, frequent system updates and code modifications introduce dynamic variations in log data, making traditional ML models struggle with unseen patterns [99]. Second, the inherent class imbalance caused by the rarity of anomalies biases models toward common patterns, often overlooking critical rare events [100]. Finally, supervised methods heavily depend on labeled datasets, which are costly and time-consuming to generate, while semi-supervised and unsupervised approaches often trade reduced labeling requirements for diminished accuracy [74, 75].

To overcome these challenges, this chapter introduces Adaptable Log-based Self-supervised method to Catch Anomalies (ALogSCAN). ALogSCAN uses a dual-network architecture comprising an Auto-Encoder (AE) teacher model and an Encoder-Only (EO) student model, leveraging Knowledge Distillation (KD) for efficient anomaly detection. Dynamic Frequency-based Log Filtering (DFLF) adapts to evolving log data by prioritizing

[1]This chapter is based on one published and one paper under second-round review: [5] Mahsa Raeiszadeh, F. Estrada-Solano, R. H. Glitho, J. Eker, and R. A. Mini, "A Data-Driven Approach for Adaptive Real-Time Log Parsing in Cloud Environments," in Proc. *IEEE International Mediterranean Conference on Communications and Networking (Meditcom)*, 2024, and [6] Mahsa Raeiszadeh, F. Estrada-Solano, R. H. Glitho, J. Eker, and R. A. Mini, "ALogSCAN: A Self-Supervised Dual Network for Adaptive and Timely Log Anomaly Detection in Clouds," Submitted to *IEEE Transactions on Machine Learning in Communications and Networking (TMLCN), 2025*

infrequent patterns, while self-supervised learning eliminates the need for labeled data by using input data as its own supervision.

The rest of this chapter is organized as follows. First, we present a motivating scenario followed by the system model and the problem formulation. Next, we present the proposed method and its components. Next, we evaluate ALogSCAN against state-of-the-art baselines and detail the experimental results. Finally, we summarize the contributions and findings.

5.2 Motivating Scenario

Since real-time clouds require timely anomaly detection to prevent service disruptions, the instability of logs introduces additional challenges for ML-based anomaly detection. Real-time cloud logs evolve dynamically and unpredictably, requiring ML models to continuously adapt without retraining delays. Frequent log changes introduce additional uncertainty, making anomaly detection even more challenging. ML models trained on historical logs often struggle to generalize to evolving log formats, leading to missed or misclassified anomalies, which can delay corrective actions.

To illustrate the impact of these challenges, consider an OpenStack-managed video streaming service that dynamically scales based on demand (Fig. 5.1). In this scenario, real-time performance is critical for ensuring seamless service delivery. Neutron must dynamically reconfigure network bandwidth in response to surges in streaming traffic. Nova must rapidly provision new virtual machines (VMs) when auto-scaling is triggered to handle increased demand. Anomaly detection must adapt instantly to changes in log patterns to prevent service degradation. During normal operation, Nova, Neutron, and Cinder generate structured logs for compute provisioning, networking, and storage tasks. However, if traffic spikes unexpectedly, OpenStack may auto-scale compute resources by launching new VMs, triggering a surge in log activity. This results in:

- New log templates from additional VMs with previously unseen status messages.
- Modified log structures as Neutron dynamically reconfigures network bandwidth.
- Deprecated error messages as Nova updates its logging mechanism for compute failures.

To quantify the impact of log instability on anomaly detection performance, we conducted an empirical study to investigate the issue of log instability using private data from the Ericsson Research Data Center (ERDC), which relies on OpenStack to deploy and manage the cloud infrastructure. The ERDC dataset covers approximately 4 million log entries collected over a three-month period. First, we took 50% of the log data and analyzed whether the log templates changed or not in the remaining 50% of data, finding that only

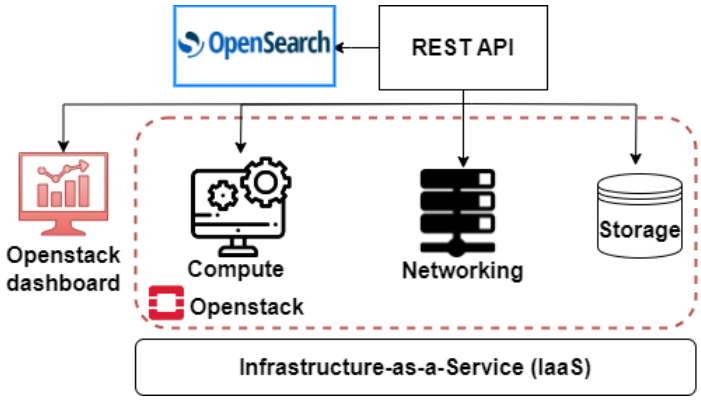
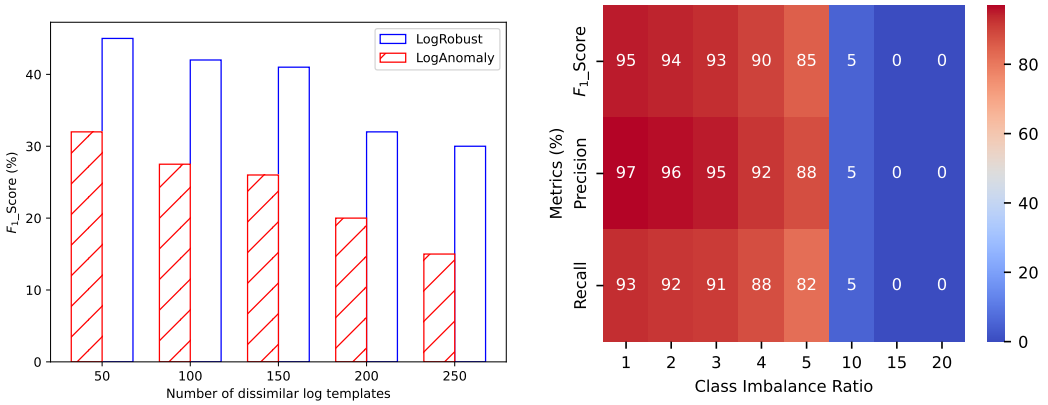


Figure 5.1: Overview of an OpenStack-managed real-time cloud infrastructure.

10.1% remained unchanged. In addition, Fig. 5.2a shows how dissimilar log templates in the ERDC dataset affect the accuracy performance of two classical anomaly detection methods: LogRobust [69] and LogAnomaly [70]. We identify dissimilar log templates from any previously observed one if the cosine similarity of their semantic vectors falls below a 0.5 threshold. Of the 1,532 unique log templates identified during the study, 720 (more than 45%) emerged as dissimilar. We trained the anomaly detection methods on 70% of the data and tested them on the remaining 30%, which was divided into five subsets to vary the number of dissimilar log templates from 50 to 250, in steps of 50.

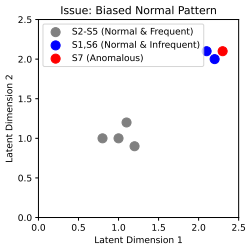


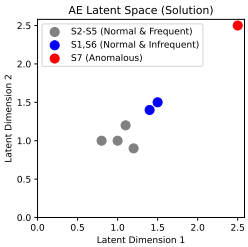
(a) (b)

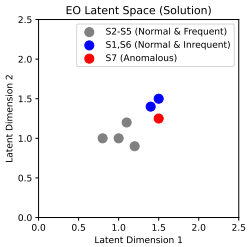
Figure 5.2: (a) Accuracy performance of two baseline anomaly detection methods while varying the number of dissimilar log templates in an OpenStack-based log dataset. (b) Accuracy performance of a two-layer LSTM while varying the class imbalance ratio in an HDFS log dataset.

The results in Fig. 5.2a reveal a gradual decrease in F_1-score of both LogRobust and LogAnomaly as the number of dissimilar log templates increases. In particular, the drop of the F_1-score is considerable when the number of dissimilar log templates is 200 (i.e., 13% of all log templates), showing that the prevalence of dissimilar log templates negatively

Sequences	Log Templates
S1	[θ1, θ2, θ3, θ4]
S2	[θ1, θ5, θ5, θ5]
S3	[θ2, θ5, θ5, θ5]
S4	[θ3, θ5, θ5, θ5]
S5	[θ4, θ5, θ5, θ5]
S6	[θ1, θ2, θ3, θ5]
S7	[θ1, θ2, θ3, θ9]







(a) (b) (c) (d)

Figure 5.3: Illustration of the biased normal pattern problem and solution. (b) shows how frequent templates can obscure anomalies. (c) and (d) demonstrate how the solution addresses the issue by contrasting networks that prioritize (c) infrequent log templates and (d) frequent log templates.

impacts the performance of anomaly detection methods. The accuracy reduction is likely due to the methods' reliance on the semantic meanings of log templates, which become less effective when many dissimilar templates exist.

Another challenge in cloud infrastructure environments is dealing with the imbalance of classes since the number of normal logs far outweighs the number of abnormal logs. For example, in the Hadoop Distributed File System (HDFS), logs capturing routine operations like writing or deleting blocks are far more frequent than rare failures in network communication or storage errors. Fig. 5.2b depicts the impact of class imbalance ratio in real-world HDFS log data [101] on the accuracy results (F_1-score, precision, and recall) of a simple two-layer LSTM classification model. Since the class imbalance ratio represents the number of normal logs divided by the number of abnormal logs, we vary the class imbalance ratio by removing abnormal logs from the dataset. The results in Fig. 5.2b show that as the class imbalance ratio increases, the accuracy performance decreases. Note the accuracy metrics dramatically decline for class imbalance ratios larger than five, dipping to zero for imbalance ratios of 20 and beyond. These results demonstrate how traditional classification models struggle to detect anomalies in highly imbalanced datasets.

Fig. 5.3 deeps into the challenge of biased normal patterns. Given a set $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{|\Theta|}\}$ of log templates generated by a parsing method [5], let $\mathcal{S} = \{S_1, S_2, \ldots, S_{|\mathcal{S}|}\}$ represent a set of log template sequences. Each $S_i \in \mathcal{S}$ describes an ordered list of $|S_i|$ log templates $S_i = [\theta_a, \ldots, \theta_b]$ $\forall \theta_a, \theta_b \in \Theta$. As depicted in Fig. 5.3a, the motivating example consists of six normal sequences, S_1 through S_6, and one anomalous sequence, S_7. Sequences S_2 to S_5 contain a common frequent log template, θ_5, whereas sequences S_1 and S_6 consist of distinct log templates. The anomalous sequence S_7 shares three log templates with S_1 (θ_1, θ_2, and θ_3) and includes an infrequent anomalous log template, θ_9.

Fig. 5.3b shows the latent space representation learned by OC-SVM, a simple anomaly detection model, for the set of log template sequences. Note the log template sequences S_2 to

S_5, consisting of frequent log templates θ_5, heavily influence the model's latent space, leading to a biased normal pattern. This representation impedes correctly distinguishing the normal sequences S_1 and S_6 from the anomalous sequence S_7, as the three share similarities and diverge from the normal pattern. In contrast, Figs. 5.3c and 5.3d overview our approach to mitigate this issue. We use a dual-network architecture, where AE focuses on the infrequent log templates and EO on the frequent ones. Note the latent space from AE (see Fig. 5.3c) differs from that from EO (see Fig. 5.3d), enabling a proper distinction between normal and abnormal log sequences.

5.3 System model

As depicted in Fig. 5.4, we consider a system model for prompt anomaly detection by processing logs originating from one or multiple services that compose the system. For example, OpenStack-based cloud infrastructures deploy different services (e.g., Nova, Neutron, and Cinder) that continuously generate log data. Log collector agents constantly capture and transmit such log data for model training and online processing.

First, for model training, log collector agents communicate historical logs to a log repository that chronologically aggregates and maintains log data, constructing an ordered list of logs $L = [l_1, \ldots, l_n]$, where each $l_i \in L$ represents a single log entry from a given service. A log parsing method, such as AdapLog [5], executes an offline process to analyze the list of historical logs and generate a set of log templates $\Theta = \{\theta_1, \ldots, \theta_{|\Theta|}\}$ representing the parsed logs. Using the set of log templates Θ, the offline log parsing builds an ordered list of log templates $\mathcal{L} = [\theta_u, \ldots, \theta_v]$ $\forall \theta_u, \theta_v \in \Theta$. Note the log templates in $\mathcal{L}$ correspond to each log in L; therefore, θ_u and θ_v represent the log templates for l_1 and l_n, respectively. Subsequently, a log sequence slicer partitions the full list of log templates $\mathcal{L}$ into sequences either by sessions or windows (see Section 5.4), generating a set of log template sequences $\mathcal{S} = \{S_1, \ldots, S_{|\mathcal{S}|}\}$. Each $S_i \in \mathcal{S}$ defines an ordered list $S_i = [\theta_a, \ldots, \theta_b]$ $\forall \theta_a, \theta_b \in \Theta$. Assuming a function $p : \theta \to \mathbb{Z}$ that returns the index position of the log template in the ordered list $\mathcal{L}$, then $p(\theta_u) \leq p(\theta_a) < p(\theta_b) \leq p(\theta_v)$. Finally, the set of log template sequences $\mathcal{S}$ feeds the training process of the anomaly detection model.

In online processing, log collector agents continuously transmit online logs l_t captured at time t to the log parsing method (e.g., AdapLog [5]). The online log-parsing process relies on the set of log templates Θ, constructed during the offline process, to produce the log template $\theta_c \in \Theta$ for each online log l_t. A log sequence arranger chronologically aggregates online log templates (either by sessions or windows) up to a quantity or a time T to build log template sequences $S_T = [\theta_c, \ldots, \theta_d]$ $\forall \theta_c, \theta_d \in \Theta$. Note the log templates in S_T correspond to the online logs, therefore, θ_c and θ_d represent the log templates for l_t and l_{t+T}, respectively. Finally, the trained anomaly detection model analyzes log template sequences S_T to identify deviations and flag potential anomalies for further investigation.

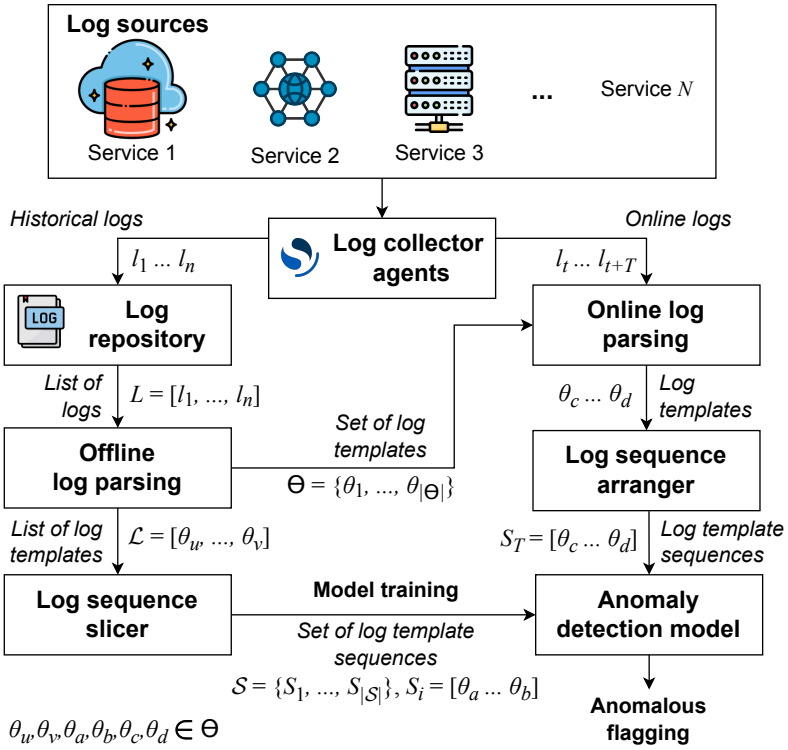


Figure 5.4: System model

5.4 Problem Formulation

Given a training set of log sequences $\mathcal{S}$, let $\mathcal{N} \subset \mathcal{S}$ be the subset of normal log sequences and $\mathcal{A} \subset \mathcal{S}$ the subset of anomalous log sequences, where $|\mathcal{N}| \gg |\mathcal{A}|$. Furthermore, $S_{\mathcal{N}} \in \mathcal{N}$ represents a normal log sequence, whereas $S_{\mathcal{A}} \in \mathcal{A}$ refers to an abnormal log sequence. Therefore, we aim to learn an anomaly scoring function $f : S_i \to \mathbb{R}$ such that for an online log sequence S_T we have $|f(S_T;\theta) - f(S_{\mathcal{N}};\theta)| > |f(S_T;\theta) - f(S_{\mathcal{A}};\theta)|$ if S_T is anomalous.

Note θ parametrizes the scoring function f as each log sequence $S_i = [\theta_a, \dots, \theta_b]$, where $\theta_a, \theta_b \in \Theta$ represent any log template and $|S_i|$ denotes the total count of log templates in the log sequence. Furthermore, each log template $\theta_a \in \Theta$ defines an ordered list of tokens $\theta_a = [w_1^a, \dots, w_{|\theta_a|}^a]$, where each token w_j^a designates a wildcard or a valid word in a pre-defined vocabulary [5]. Using a word embedding technique [3], we map each token w_j^a as a vector representation $x_j^a \in \mathbb{R}^m$ to enable learning the semantic relationships across tokens.

In this thesis, template sequences refer to the groups of standard log templates of chronologically consecutive log messages that capture the system execution flow. Template sequences are partitioned either by sessions or windows. Session-based partitioning relies on log identifiers to form sequences of related operations. Window-based partitioning defines a fixed time or entry amount. Fixed time windows group logs within consistent time frames (e.g., 15 minutes, 1 hour); therefore, sequence lengths vary depending on the system workload. Fixed entry windows set an entry number (e.g., 60, 100), providing uniform

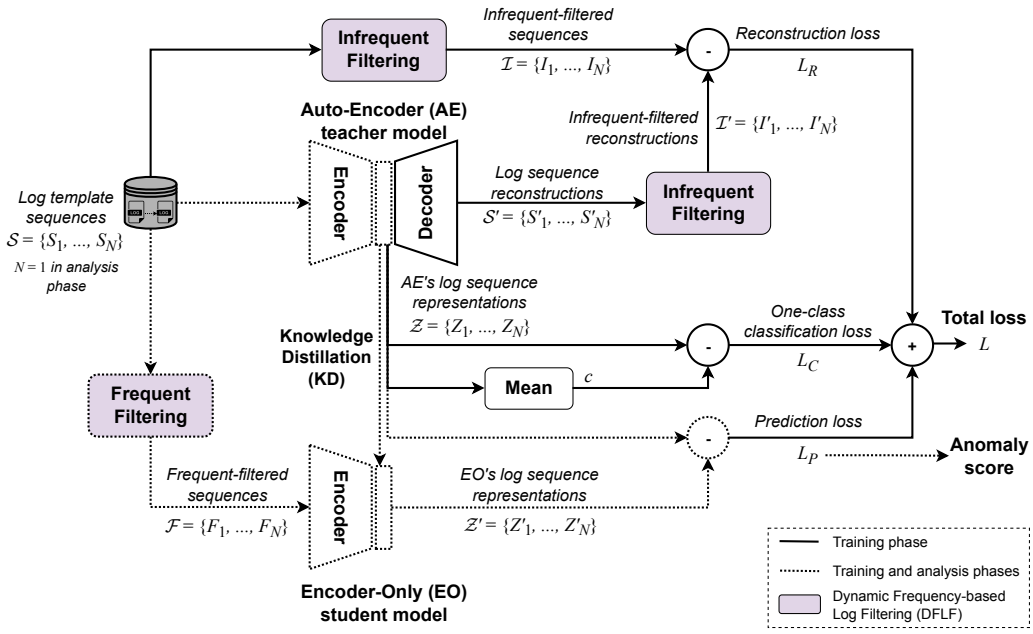


Figure 5.5: ALogSCAN architecture

sequence lengths.

5.5 Proposed Solution

Fig. 5.5 introduces ALogSCAN, a log anomaly detection method that leverages self-supervised learning, KD, and frequency filtering techniques to operate with unlabeled, unstable, and imbalanced log data and accurately and promptly identify anomalous log sequences. ALogSCAN implements a dual-network architecture: an AE teacher model and an EO student model. The AE teacher model operates with complete log sequences and focuses on reconstructing infrequent log templates, whereas the EO student model handles sequences with frequent log templates. ALogSCAN distills knowledge from the AE teacher model to the EO student model and introduces a DFLF technique that dynamically categorizes infrequent and frequent log templates in log sequences. This dual-network architecture operates in two phases: training and analysis. Error losses from the AE's infrequent-based reconstruction and the representations of the two models (i.e., AE and EO) guide the training phase. In the analysis phase, representation errors between the AE teacher and EO student models enable ALogSCAN to timely detect anomalous log sequences. Algorithms 5.1 and 5.2 depict the training and analysis phases, respectively.

Algorithm 5.1: ALogSCAN's training phase

input : set of log sequences $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$
output: updated Auto-Encoder AE and Encoder-Only EO

1 **begin** on processing $\mathcal{S}$
  `// Log template filtering`
2 $\mathcal{I} \leftarrow DFLF.\text{Infrequent\_Filtering}(\mathcal{S})$;
3 $\mathcal{F} \leftarrow DFLF.\text{Frequent\_Filtering}(\mathcal{S})$;
  `// AE model`
4 $\mathcal{Z} \leftarrow AE.enconder.\text{Latent\_Representation}(\mathcal{S})$;
5 $\mathcal{S}' \leftarrow AE.decoder.\text{Reconstruction}(\mathcal{Z})$;
6 $\mathcal{I}' \leftarrow DFLF.\text{Infrequent\_Filtering}(\mathcal{S}')$;
7 $L_R \leftarrow \text{Reconstruction\_Loss}(\mathcal{I}, \mathcal{I}')$;
8 $L_C \leftarrow \text{Oneclass\_Classification\_Loss}(\mathcal{Z})$;
  `// EO model`
9 $\mathcal{Z}' \leftarrow EO.\text{Latent\_Representation}(\mathcal{F})$;
10 $L_P \leftarrow \text{Prediction\_Loss}(\mathcal{Z}, \mathcal{Z}')$;
  `// Models update`
11 $L \leftarrow \text{Total\_Loss}(L_R, L_C, L_P)$;
12 **update** $AE \leftarrow AE.\text{Backpropagation}(L)$;
13 **update** $EO \leftarrow EO.\text{Backpropagation}(L)$;
14 **update** $EO.encoder \leftarrow AE.encoder.\text{KD}()$;
15 **return** AE **and** EO;

5.5.1 Training phase

As illustrated in Fig. 5.5, ALogSCAN's training phase involves the complete dual-network architecture. Algorithm 5.1 details ALogSCAN's training phase. The training phase executes multiple epochs and operates with batches that divide the full training set of log sequences. For simplicity, we denote a batch of N log sequences as the set $\mathcal{S} = \{S_1, \ldots, S_N\}$ since the training process works the same at each epoch and for each batch.

As shown in lines 2-3 in Algorithm 5.1, ALogSCAN uses DFLF to perform infrequent and frequent filtering on each $S^i \in \mathcal{S}$ to generate the corresponding sets of filtered sequences $\mathcal{I} = \{I_1, \ldots, I_N\}$ and $\mathcal{F} = \{F_1, \ldots, L_N\}$. $I_i \in \mathcal{I}$ preserves the infrequent log templates in log sequences $S^i \in \mathcal{S}$, whereas the frequent ones remain in $F_i \in \mathcal{F}$. DFLF dynamically adjusts the filtering frequency ratio, as detailed in Section 5.5.3.

Considering the example in Fig 5.3a, infrequent filtering removes the frequent log template θ_5 and preserves the infrequent log templates θ_1 to θ_4 and θ_9. Therefore, the infrequent-filtered log sequences I_i modify the log sequences S_2 to S_6 (e.g., $I_2 = [\theta_1, 0, 0, 0]$), whereas I_1 and I_7 contain the same log templates as S_1 and S_7, respectively. In contrast, frequent filtering retains the frequent log template θ_5 and discards the infrequent log templates θ_1 to θ_4 and θ_9. Therefore, the frequent-filtered log sequences F_i change for all the log sequences S_1 to S_7 (e.g., $F_i = [0, \theta_5, \theta_5, \theta_5]$ $\forall i \in [2-5]$).

As depicted in lines 4-5 in Algorithm 5.1, the AE teacher model operates with the unfiltered log sequences $S_i \in \mathcal{S}$ to produce a set of latent representations $\mathcal{Z} = \{Z_1, \ldots, Z_N\}$ from its encoder function (a.k.a., bottleneck layer) and the corresponding set of log sequence

reconstructions $\mathcal{S}' = \{S'_1, \ldots, S'_N\}$ from its decoder function. Subsequently, ALogSCAN performs infrequent filtering (i.e., DFLF) on the reconstructions $S'_i \in \mathcal{S}'$ to generate the set of infrequent-filtered log sequence reconstructions $\mathcal{I}' = \{I'_1, \ldots, I'_N\}$ (Algorithm 5.1, line 6). Like filtered log sequences, $I'_i \in \mathcal{I}'$ preserves the infrequent log templates in the log sequence reconstructions $S'_i \in \mathcal{S}'$.

ALogSCAN uses the infrequent-filtered sets of log sequences ($\mathcal{I}$) and of log sequence reconstructions ($\mathcal{I}'$) to compute a reconstruction loss L_R (Algorithm 5.1, line 7). L_R drives the AE teacher model to improve the reconstruction of the infrequent templates in the filtered log sequences $I_i \in \mathcal{I}$, as described in Section 5.5.4. Consequently, the AE teacher model develops latent representations $Z_i \in \mathcal{Z}$ that emphasize the infrequent templates within the input log sequences $S_i \in \mathcal{S}$. As shown in line 8 in Algorithm 5.1, ALogSCAN calculates a one-class classification loss L_C from the deviations of the latent representations $Z_i \in \mathcal{Z}$ from its mean $\bar{Z}$. L_C promotes the clustering of latent representations in the AE teacher model. Section 5.5.6 details the equations to compute the reconstruction (L_R) and classification (L_C) losses.

As described in line 9 in Algorithm 5.1, the EO student model works with frequent-filtered log sequences $F_i \in \mathcal{F}$ to produce a set of latent representations $\mathcal{Z}' = \{Z'_1, \ldots, Z'_N\}$. Considering that the EO student model's design reflects the AE teacher model's encoder function and that its input data $\mathcal{F}$ disregards infrequent templates, EO's encoder function learns the latent representations of frequent log templates within a log sequence $S_i \in \mathcal{S}$. Afterward, ALogSCAN computes a prediction loss L_P by comparing the latent representations $Z_i \in \mathcal{Z}$ and $Z'_i \in \mathcal{Z}'$ generated by the AE teacher and EO student models, respectively (Algorithm 5.1, line 10). L_P optimizes the similarity between the latent representations to ensure they align for normal log sequences. ALogSCAN combines the reconstruction (L_R), classification (L_C), and prediction (L_P) losses to construct the total loss L (Algorithm 5.1, line 11). Section 5.5.6 details the equations to compute the prediction (L_P) and total (L) losses. Finally, as denoted in lines 12-14 in Algorithm 5.1, the AE teacher and EO student models execute backpropagation using the total loss function L to update their model parameters, and the AE teacher model distills knowledge to the EO student model. Section 5.5.5 further describes the KD technique.

5.5.2 Analysis phase

As depicted in Fig. 5.5, ALogSCAN's analysis phase involves only the encoder from the AE teacher model and the EO student model. Algorithm 5.2 describes ALogSCAN's analysis phase. In contrast to the training phase, the analysis phase operates with an online log sequence S at a time for prompt anomaly detection. First, ALogSCAN only performs DFLF frequent filtering on S to generate the corresponding filtered log sequence F (Algorithm 5.2, line 2). Note F preserves the frequent log template from the log sequence S.

As shown in lines 3-4 in Algorithm 5.2, the AE teacher model' encoder uses the log

Algorithm 5.2: ALogSCAN's analysis phase

input : online log sequence S, trained Auto-Encoder AE, trained Encoder-Only EO
output: anomaly score L_P

1 **begin** on processing S
    `// Log template filtering`
2 $F \leftarrow DFLF.\text{Frequent\_Filtering}(S)$;
    `// AE model`
3 $Z \leftarrow AE.enconder.\text{Latent\_Representation}(S)$;
    `// EO model`
4 $Z' \leftarrow EO.\text{Latent\_Representation}(F)$;
    `// Anomaly score`
5 $L_P \leftarrow \text{Prediction\_Loss}(Z, Z')$;
6 **return** L_P

sequence S to produce the latent representation Z, whereas the EO student model employs the frequent-filtered log sequence F to generate the latent representation Z'. ALogSCAN then computes the prediction loss L_P from the difference between the latent representations Z and Z' (Algorithm 5.2, line 5). L_P denotes the anomaly score indicating the abnormality of the online log sequence S (Algorithm 5.2, line 6). If the latent representations deviate significantly from the normal data distribution, the log sequence S gains a high anomaly score.

Let's consider the example in Fig 5.3a. For a normal sequence like S_6, the representations Z_6 and Z'_6 from the AE teacher and EO student models, respectively, align closely with the normal pattern in the latent space, as shown in Figs. 5.3c and 5.3d. Consequently, the difference between these two latent representations (i.e., the anomaly score L_P)) is low. Conversely, for an abnormal sequence like S_7, the representations Z_7 and Z'_7 diverge. The EO student model's representation Z'_7 remains close to the normal pattern since the infrequent (and abnormal) template θ_9 was masked to zero during the DFLF frequent filtering. However, the AE teacher model's representation Z_7 deviates from the normal pattern due to its inability to reconstruct the abnormal template θ_9. This divergence results in a high anomaly score L_P for S_7, enabling the prompt detection of the anomalous log sequence.

5.5.3 Dynamic Frequency-based Log Filtering

DFLF introduces a filtering technique based on log template frequency. In ALogSCAN, DFLF calculates template occurrence frequencies for each batch of log sequences to define a frequency rate threshold κ. DFLF categorizes log templates as infrequent and frequent if their frequency rate is less than and more than κ, respectively. The frequency threshold κ enables establishing either a fixed value or a ratio of unique templates within a batch, serving as a hyperparameter. In particular, ALogSCAN dynamically selects a unique filtering ratio for each batch. We randomly sample the frequency threshold κ from a set $K = \{k_1, \dots k_{|K|}\}$,

where each $k_i \in K$ represents a possible value for κ. This sampling provides randomization during the DFLF process to promote variability and robustness during training, helping the model to focus on infrequent events.

DFLF defines two filtering processes: infrequent and frequent. Infrequent filtering preserves infrequent log templates and removes the frequent ones from the log sequences. In contrast, frequent filtering maintains frequent log templates and discards the infrequent ones from the log sequences. As illustrated in Fig. 5.5, ALogSCAN applies infrequent filtering to the input log sequences $\mathcal{S}$ and AE's reconstructions $\mathcal{S}'$ for computing the reconstruction loss L_R. This approach enables the calculation of the reconstruction loss based on the infrequent log templates, driving the AE teacher model to learn their reconstruction. Conversely, ALogSCAN implements frequent filtering on the input log sequences $\mathcal{S}$ to feed the EO student model. This frequent-filtered input $\mathcal{F}$ allows EO to learn representations of $\mathcal{S}$ without information from the infrequent log templates. Leveraging this DFLF combination of infrequent and frequent filtering, ALogSCAN effectively learns less biased normal patterns in the latent representations of the AE teacher and EO student models, prioritizing infrequent log templates.

5.5.4 Frequency-based reconstruction

The AE teacher model in ALogSCAN relies on DFLF to perform reconstruction based on the frequency of log templates. This frequency-based reconstruction differs from the traditional Reconstruction with Complete Input (RCI). As shown in Fig. 5.6a, RCI utilizes the entire input of log sequences to feed the AE model and computes the reconstruction loss by comparing all log templates in the original and reconstructed log sequences.

In contrast, frequency-based reconstruction introduces two techniques: Infrequent-based Reconstruction with Frequent Input (IRFI) and Infrequent-based Reconstruction with Complete Input (IRCI). As depicted in Figs. 5.6b and 5.6c, both IRFI and IRCI calculate the reconstruction loss by comparing only the infrequent log templates from the original and reconstructed log sequences. In particular, IRFI preserves only the frequent log templates in the input of log sequences, guiding the AE model to focus on frequent events. In contrast, IRCI uses all the log templates in log sequences to feed the AE model, leveraging their entire range to reconstruct infrequent log templates accurately. Consequently, the AE model prioritizes the reconstruction of infrequent log templates rather than attempting to rebuild the complete log sequence. ALogSCAN relies on IRCI to improve the anomaly detection accuracy, as demonstrated in Section 5.6.2.

5.5.5 Knowledge Distillation

After the AE teacher model trains the encoder, it can transform input data into latent representations that capture important patterns and features. Therefore, the EO student model leverages KD and deploys only the AE's encoder for anomaly detection instead

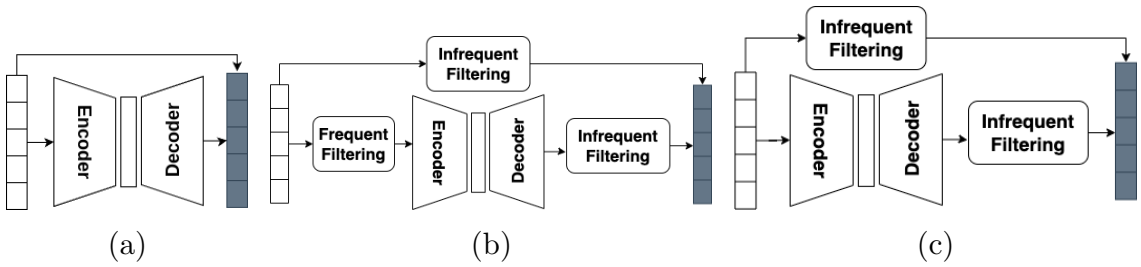


Figure 5.6: Auto-Encoder (AE) reconstruction techniques: (a) Reconstruction with Complete Input (RCI), (b) Infrequent-based Reconstruction with Frequent Input, and (c) Infrequent-based Reconstruction with Complete Input (IRCI).

of using the full AE teacher model, which would require reconstruction errors to detect anomalies. During the training phase, the AE teacher model learns from the encoder and decoder, whereas the EO student model focuses only on the encoder's output. Leveraging KD, the EO student model learns to compress data into meaningful latent representations, aiming to match those generated by the AE teacher model's encoder. During the analysis phase, ALogSCAN identifies an anomaly when an EO's latent representation fails to align well with the AE's latent representation. We add a classifier to the EO student model that operates on the latent representations to predict whether the log sequence represents an anomaly directly.

5.5.6 Self-supervised learning

ALogSCAN defines a dual-network architecture incorporating self-supervised learning to guide model optimization effectively. The dual-network architecture consists of two neural network models: AE and EO. Both models use Convolutional Neural Networks (CNN) to implement the corresponding encoders and decoders. To optimize the models during the training phase, ALogSCAN defines a total loss function L that combines three losses from self-supervised learning techniques: reconstruction loss (L_R), one-class classification loss (L_C), and prediction loss (L_P).

Reconstruction loss (L_R): Following the frequency-based reconstruction technique (see Section 5.5.4), ALogSCAN uses the infrequent-filtered log sequences $I_i \in \mathcal{I}$ and their corresponding infrequent-filtered reconstructions $I'_i \in \mathcal{I}'$ to compute the reconstruction loss L_R (see Equation 5.1). Since $\mathcal{I}$ and $\mathcal{I}'$ derive from infrequent filtering (see Section 5.5.3), the reconstruction loss L_R focuses on infrequent log templates in the log sequences $S_i \in \mathcal{S}$. We employ the Mean Square Error (MSE) to calculate the reconstruction loss L_R as follows:

$$L_R = \frac{1}{N * d} \sum_{i=1}^{N} \frac{1}{M_i} \sum_{j=1}^{M_i} \left\| x_{ij} - x'_{ij} \right\|_2^2 \tag{5.1}$$

where $x_{ij}, x'_{ij} \in \mathbb{R}^d$ denote the d-dimensional semantic features of the j-th log template in

the i-th log sequence of the set of infrequent-filtered log sequences $\mathcal{I}$ and reconstructions $\mathcal{I}'$ (i.e., I_i and I'_i, respectively). In addition, M_i is the total number of infrequent log templates in the given log sequence $S_i \in \mathcal{S}$, N is the total number of log sequences in the batch $\mathcal{S}$, and $\|\cdot\|_2^2$ refers to the squared L_2 norm.

One-class classification loss (L_C): We adopt a one-class classification technique to govern the distribution of learned representations in the latent space generated by the AE's encoder. To facilitate the convergence of representations for normal log sequences in the training set, ALogSCAN continually updates the central representation c by computing the mean of the generated representations in each epoch. Consequently, the training process aims to minimize the one-class classification loss L_C. This leads the AE's encoder to produce latent representations of normal log sequences that closely align with the center c, and latent representations of abnormal log sequences that exhibit a substantial distance from this central reference point. In addition, L_C benefits the convergence of the reconstruction loss L_R by shrinking the space that contains the log sequences' latent representations. The one-class classification loss L_C is defined as follows:

$$L_C = \frac{1}{N} \sum_{i=1}^{N} \|Z_i - c\|_2^2 \tag{5.2}$$

where $Z_i \in \mathbb{R}^l$ denotes the l-dimensional latent representation generated by the AE's encoder for the i-th log sequence (i.e., $S_i \in \mathcal{S}$), c represents the center representation of log sequences, calculated as $c = \frac{1}{N} \sum_{i=1}^{N} Z_i$, and N is the total number of log sequences in the batch $\mathcal{S}$.

Prediction loss (L_P): In alignment with KD principles, ALogSCAN utilizes the AE model's encoder to process entire log sequences $S_i \in \mathcal{S}$, generating latent representations $Z_i \in \mathcal{Z}$. The EO student model distills knowledge from the AE teacher model and processes frequent-filtered log sequences $F_i \in \mathcal{F}$ to generate latent representations $Z'_i \in \mathcal{Z}'$. This setup enables the AE teacher model to gain more insights into normal patterns than the EO student model. Therefore, EO's latent representations $Z'_i \in \mathcal{Z}'$ aim at predicting AE's latent representations $Z_i \in \mathcal{Z}$. We use MSE to compute the prediction loss L_P between EO and AE's latent representations, as follows:

$$L_p = \frac{1}{|N| * |l|} \sum_{i=1}^{N} \|Z_i - Z'_i\|_2^2 \tag{5.3}$$

where $Z_i, Z'_i \in \mathbb{R}^l$ denote the l-dimensional latent representation of the i-th log sequence (i.e., $S_i \in \mathcal{S}$) generated by the AE's encoder and EO models, and N is the total number of log sequences in the batch $\mathcal{S}$. Note $N = 1$ in the analysis phase since ALogSCAN processes one log sequence S at a time.

Finally, Eq. (5.4) defines the objective loss function L for training ALogSCAN. This loss function L includes a hyper-parameter α that balances the reconstruction loss L_R with the

other two losses: one-class classification loss L_C and prediction loss L_P. In particular, α controls the relative contribution of L_R compared to the other terms in the latent space.

$$L = \alpha L_R + L_C + L_P \tag{5.4}$$

5.6 Performance Evaluation

This section presents a performance evaluation of the proposed ALogSCAN method in terms of accuracy and time. We describe the datasets, the prototype implementation, the baseline methods, the evaluation metrics, and the findings.

5.6.1 Experiment Settings

5.6.1.1 Datasets

Our evaluation employed two public log datasets, HDFS and BGL, and a private log dataset, ERDC. The public datasets [101] have been widely recognized in log analysis research due to their real-world origin and comprehensive labeling. In particular, the HDFS dataset collects logs of executing Hadoop-based MapReduce jobs on over 200 Amazon EC2 nodes. HDFS logs were manually labeled via handcrafted rules to identify anomalies. The BGL dataset provides logs from the BlueGene/L supercomputer, equipped with 131,072 processors and 32,768 GB of memory. BGL logs contain alert and non-alert messages identified by alert category labels.

The private ERDC dataset collects operational log data from hardware and software systems within the Ericsson Research Data Center in Lund, Sweden. The log data originates from Xerces, a private cloud service deployed on ERDC. The ERDC infrastructure involves approximately 300 servers orchestrated by the OpenStack cloud platform. The logs cover various OpenStack subsystems, each responsible for specific cloud functions, including Nova, Neutron, Swift, Cinder, Horizon, and Keystone.

We employed a chronological approach and followed a 60/20% batch decomposition for dividing the datasets into training and test datasets. Furthermore, we grouped logs into sequences using either session- or window-based partitioning. In HDFS, we applied session-based partitioning to group logs by BlockID. In BGL and ERDC, we used fixed entry windows of 20, 60, and 100 logs to evaluate the balance between detection performance and storage efficiency. We preferred fixed entry window partitioning over fixed time windows since the former provides uniform sequence lengths.

5.6.1.2 Prototype implementation

Log template sequences represent ALogSCAN's input data (see Fig. 5.5). We rely on AdapLog [5] for building the log template sequences generated by the offline and online log

parsing processes (see Fig. 5.4). Regarding frequent and infrequent filtering of log template sequences, our DFLF implementation dynamically selects the frequency threshold κ from the set $K = \{0.05, 0.1, 0.15, 0.2, 0.3\}$.

We used convolution modules from PyTorch 1.12.0 [102] to implement ALogSCAN's dual-network architecture (i.e., AE and EO). Our model implementation extracts 32-dimensional semantic embeddings from each log template sequence. The encoder model in AE and EO deploys a 2-dimensional (2D) convolution layer that employs the semantic embeddings as input data with 1 channel and generates data representations with 128 channels. The AE's decoder model implements a 2D transposed convolution layer that receives the encoder's data representations with 128 channels as input and produces data reconstructions with 1 channel. We varied the kernel size of both convolution layers among [3, 4, 5] to find the best feature extraction. The results presented in this thesis (see Section 5.6.2) use a convolving kernel of 4.

The training process of the encoder and decoder models uses AdamW optimizer, with a polynomial learning rate schedule starting at 1×10^{-2} and decaying to 1×10^{-4}. In addition, model training implements a mini-batch size of 64 and runs for up to 100 epochs, with early stopping if no improvement is observed over 20 consecutive iterations. Finally, after several tests, we fixed the reconstruction loss weight α to 100 as it provided the best balance between the loss terms in the objective loss function (see Equation 5.4).

5.6.1.3 Baseline methods

We compare ALogSCAN with six state-of-the-art log anomaly detection methods: LogRobust [69], AE+IF [72], LogAnomaly [70], DeepLog [73], PLElog [74], and LogBERT [75]. As described in Section 2.3.4, these methods involve different neural network architectures, including GRU, LSTM, BiLSTM, Transformer, and CNN models. The implementations of the selected methods are publicly available in code repositories [101], facilitating reproducibility and consistency in our evaluation. We adhered to the original configuration parameters established by the authors in their implementations. Each log anomaly detection method, including ALogSCAN, was executed three times on each dataset. This thesis reports the average results between the executions to ensure robustness. Finally, we conducted all the experiments on a Linux server running Ubuntu 20.04, equipped with an AMD Ryzen 3.5GHz CPU and 32 GB RAM.

5.6.1.4 Evaluation metrics

We evaluate the accuracy performance of ALogSCAN and the baseline anomaly detection methods using metrics derived from the confusion matrix: precision, recall, F_1-score, and the Matthews Correlation Coefficient (MCC). Considering anomalies as the positive condition, these metrics provide a comprehensive assessment of each method's ability to correctly

detect True Positives (TP) while minimizing False Positives (FP) and False Negatives (FN). See Section 4.6.1.6 for the corresponding formula.

MCC evaluates the overall prediction quality by considering all the values from the confusion matrix to provide a measure ranging from -1 to +1 (see Eq. 5.5). As MCC approaches +1, the log anomaly detection method becomes more accurate. MCC values between 0 and -1 indicate that the method is less accurate than random guessing.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.5}$$

In addition, we measure the training and analysis time of the anomaly detection methods to evaluate their timely detection capabilities. The training time monitors the time to train the ML model until convergence. The analysis time measures the time required to process online log sequences to detect anomalies (if any). The shorter the analysis time, the better the anomaly detection method for taking preventive actions.

Finally, we introduced an instability ratio to evaluate the adaptability of the anomaly detection methods. The instability ratio indicates the proportion of log sequences containing synthetic modifications (e.g., random deletions, repetitions, or shuffling). We measured the change in F_1-score per different instability ratio values to assess how well the anomaly detection methods adapt to real-world log instability. A comparison of the accuracy performance at various levels of instability enables evaluating the methods' robustness and adaptability to real-world log data where instability is present.

5.6.2 Evaluation Results

This section discusses our evaluation results broken into three parts. First, a performance evaluation of ALogSCAN and the baseline anomaly detection methods regarding accuracy, time, and adaptability. Second, an ablation study of the reconstruction and filtering techniques of ALogSCAN. Third, we assess the impact of ALogSCAN configuration parameters on the accuracy performance.

5.6.2.1 Anomaly detection performance

(A) Accuracy: Table 5.1 presents the accuracy performance comparison of ALogSCAN and six baseline methods across HDFS (session-based), BGL (100-log window), and ERDC (100-log window) datasets. The results show that ALogSCAN consistently achieved the highest MCC, F_1-score, precision, and recall across all datasets. In the HDFS dataset, ALogSCAN achieved an F_1-score of 95.83%, surpassing all baseline methods. PLELog performed comparably well with an F_1-score of 94.55%, while LogRobust ranked second overall. LogAnomaly exhibited the highest precision (99.85%) but suffered from a significantly lower recall (58.56%), indicating a high false-negative rate. For the BGL dataset, ALogSCAN significantly outperformed all baselines, achieving an MCC of 0.9483

Table 5.1: Accuracy performance of ALogSCAN and the baseline anomaly detection methods on HDFS, 100-log BGL, and 100-log ERDC datasets.

Dataset	Metric	Methods						
		LogRobust [69]	**AE+IF [72]**	**LogAnomaly [70]**	**DeepLog [73]**	**PLELog [74]**	**LogBERT [75]**	**ALogSCAN**
HDFS	MCC	0.9156	0.7570	0.7524	0.9393	<u>0.9447</u>	0.7479	**0.9559**
	F_1-score	<u>94.55</u>	74.94	73.82	90.17	95.83	72.56	**96.32**
	Precision	94.05	84.45	**99.85**	<u>97.16</u>	96.17	94.35	95.87
	Recall	90.54	67.35	58.56	84.11	<u>92.99</u>	58.94	**95.80**
BGL	MCC	0.6827	0.5761	0.6320	0.6059	<u>0.8122</u>	0.7419	**0.9483**
	F_1-score	71.91	61.92	64.05	63.64	<u>82.51</u>	76.56	**96.32**
	Precision	66.30	74.50	69.52	67.67	74.41	<u>84.08</u>	**96.00**
	Recall	78.55	52.97	59.38	60.06	<u>92.59</u>	70.28	**96.64**
ERDC	MCC	0.3331	0.4001	0.4412	0.3975	<u>0.5283</u>	0.4117	**0.8957**
	F_1-score	43.87	36.51	46.21	46.94	<u>54.96</u>	42.66	**89.74**
	Precision	36.48	30.53	42.91	40.81	<u>43.22</u>	32.36	**83.86**
	Recall	55.02	45.41	50.07	55.23	<u>75.44</u>	62.59	**96.50**

The best and second-best accuracy results are in bold and underlined, respectively.

and an F_1-score of 96.32%. PLELog ranked second with an MCC of 0.8122 and an F_1-score of 82.51%, showing strong recall (92.59%) but lower precision (74.41%) compared to LogBERT (84.08%). In the ERDC dataset, ALogSCAN achieved the highest gains, with an MCC of 0.8957 and an F_1-score of 89.74%. PLELog was the second-best method, with an MCC of 0.5283 and an F_1-score of 54.96%.

Regarding the baseline methods, PLELog ranked as the second-best accurate method across most datasets, indicating that combining probabilistic label estimation and semi-supervised learning is highly effective for log anomaly detection. LogBERT performed well on the BGL and ERDC datasets, outperforming other methods such as LogRobust, AE+IF, Log LogAnomaly, and DeepLog. However, LogBERT still falls short of ALogSCAN's accuracy performance across all metrics and datasets. AE+IF exhibited a weak performance, particularly on the ERDC dataset, where it failed to correctly distinguish anomalies from normal log sequences. This suggests that global reconstruction approaches struggle to capture subtle log patterns in large-scale cloud environments.

In conclusion, the results show that ALogSCAN's CNN-based architecture outperformed all other methods based on neural networks, including BiLSTM (AE+IF), LSTM (PLELog, DeepLog, LogAnomaly), and Transformers (LogRobust, LogBERT). Note that Transformer-based methods like LogRobust and LogBERT generally performed better than LSTM-based methods such as DeepLog and LogAnomaly. Nevertheless, PLELog (an LSTM-based method) achieved better results than the Transformer-based counterparts in most metrics and datasets. This behavior demonstrates that, in addition to the neural network architecture, the underlying anomaly detection techniques (e.g., reconstruction, one-class classification, and probabilistic label estimation) play a critical role in detection performance.

(B) Time efficiency: We evaluated the time efficiency of ALogSCAN and the six baseline methods during model training and log sequence analysis. The training time measures the time the ML model takes to converge. The analysis time computes the time to

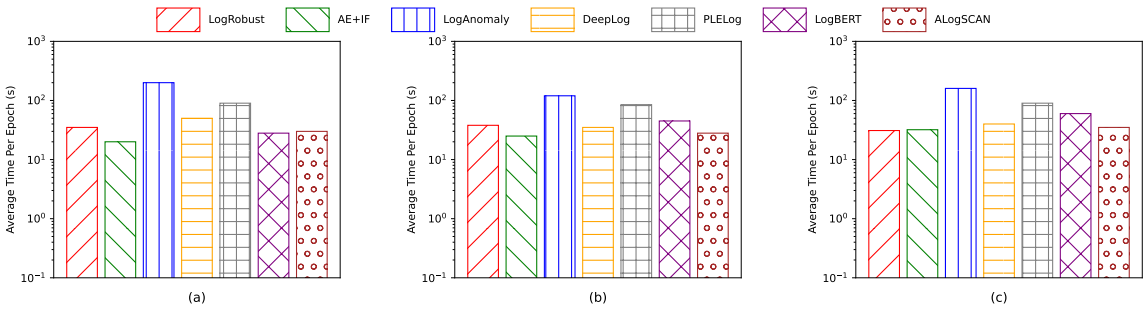


Figure 5.7: Training time of anomaly detection methods in (a) HDFS, (b) 100-log window BGL, and (c) 100-log window ERDC datasets.

process online log sequences and detect anomalies (if any). We conducted the experiments to evaluate the time efficiency using three datasets: session-based HDFS and 100-log window BGL and ERDC.

Fig. 5.7 depicts each anomaly detection method's average training time per epoch. The results reveal that LogAnomaly and PLELog consistently exhibited the longest training times, displaying the highest computational demands during training. LogAnomaly reached a peak training time of 200 seconds per epoch in the HDFS dataset, reflecting its heavy computational requirements when processing extensive logs during model training. Similarly, PLELog maintained approximately 90 seconds per epoch across all datasets. This consistent overhead reflects PLELog's complexity in handling large log sequences, leading to significant computational costs.

LogBERT and DeepLog reduced the training time of the two previous methods but their performance varied depending on the dataset. LogBERT achieved a short training time in HDFS (39 seconds) but increased in the other two datasets, reaching up to 60 seconds per epoch in ERDC. This increment is due to the complexity of LogBERT's random masking and sequence prediction mechanisms. In contrast, DeepLog produced a short training time in BGL and ERDC datasets (39 seconds on average) but incremented to 50 seconds per epoch in HDFS. This increase in a large dataset such as HDFS is due to DeepLog's need to break longer sequences into smaller segments during training to maintain performance.

The training time of LogRobust, ALogSCAN, and AE+IF remained consistently short across the three datasets, with an average value of 36, 32, and 26 seconds per epoch, respectively. Note that AE+IF exhibited superior training efficiency, highlighting its lightweight architecture. However, as shown in Table 5.1, this lightweight architecture produced poor anomaly detection in the three datasets. In contrast, ALogSCAN ranked as the second-fastest method in training time and provided the most accurate performance, demonstrating the advantage of our proposal in handling large-scale log datasets with minimal computational overhead during training.

Continuing with the time efficiency evaluation, Fig. 5.8 shows the average analysis time

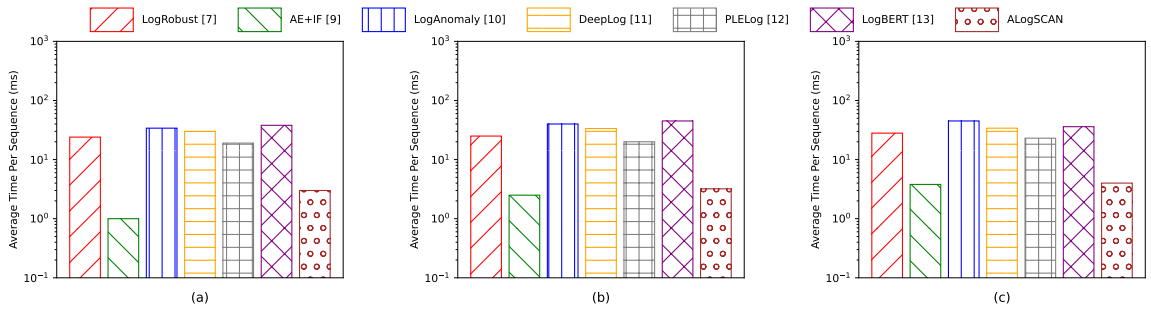


Figure 5.8: Analysis time of anomaly detection methods in (a) HDFS, (b) 100-log window BGL, and (c) 100-log window ERDC datasets.

per log sequence each anomaly detection method takes. The analysis time is measured in milliseconds (ms) and reflects the method's efficiency for timely detection. The results indicate that LogAnomaly and LogBERT performed the slowest analysis, both requiring between 34 and 45 ms per log sequence across the three datasets. These values reflect the complexity of LogAnomaly and LogBERT's prediction mechanisms, making them unsuitable for strict time conditions. DeepLog and LogRobust produced minor and more consistent values than the previous methods, requiring an average analysis time of 32 and 25 ms, respectively. However, these two methods still present challenges for prompt anomaly detection. Similarly, PLELog maintained a consistent analysis time across the three datasets, taking around 19, 20 and 23 ms to analyze each log sequence. Although PLELog performed faster than the previous methods, it is still far from the fastest analysis times of AE+IF and ALogSCAN.

AE+IF produced an excellent analysis time in the HDFS dataset, requiring only 1 ms per log sequence. Considering that AE+IF's accuracy performance in HDFS was acceptable (i.e., F_1-score of 74.94% as shown in Table 5.1), this time value indicates its suitability for systems operating with data similar to HDFS and requiring rapid detection and quick response times. Nevertheless, if the requirement is high accuracy and timely anomaly detection, the results suggest that ALogSCAN is the best method. In HDFS, ALogSCAN achieved the best accuracy (i.e., F_1-score of 96.32%) while taking only 2 ms more than AE+IF to analyze each log sequence. In fact, ALogSCAN maintained this outstanding performance in terms of accuracy and analysis time in the other two datasets. ALogSCAN provided the most accurate anomaly detection method in the BGL and ERDC datasets and requires an analysis time of only 3 and 4 ms per log sequence, respectively. These time values are comparable and less than the analysis time produced by AE+IF in BGL and ERDC, respectively. Therefore, the results demonstrate ALogSCAN's capacity to detect anomalies with minimal delay.

Table 5.2: Steps to generate unstable log sequences.

Original log sequence	$\theta_1 \rightarrow \theta_2 \rightarrow \theta_3 \rightarrow \theta_4 \rightarrow \theta_5$
Deletion in sequence	$\theta_1 \rightarrow \theta_2 \rightarrow \theta_3 \rightarrow \theta_4 \rightarrow \theta_5$
Shuffle in sequence	$\theta_1 \rightarrow \theta_4 \rightarrow \theta_2 \rightarrow \theta_3 \rightarrow \theta_5$
Insertion in sequence	$\theta_1 \rightarrow \theta_2 \rightarrow \theta_3 \rightarrow \theta_6 \rightarrow \theta_4 \rightarrow \theta_5$

(C) Adaptability: We conducted two experiments to evaluate the adaptability of the anomaly detection methods in dynamic environments: (i) log instability via synthetic log injections and (ii) the impact of anomalies in the test data. This evaluation compared ALogSCAN with LogRobust [69] and DeepLog [73] in the HDFS dataset. We analyzed LogRobust as it addresses log instability and handles variations in log patterns. DeepLog, on the other hand, provides a log key-based approach, serving as a valuable representation of traditional methods.

For the first experiment, we generated unstable log sequence data as depicted in Table 5.2. We followed four steps to replicate unstable log sequences. First, we randomly removed a few log templates from the original sequences. Second, we shuffled sub-sequences within the log templates. Third, we randomly chose a non-critical log template and repeated it multiple times within the sequence. Finally, synthetic logs were then randomly inserted into the original HDFS dataset. We trained the anomaly detection methods using the original dataset and then tested them on the modified data, including the injected synthetic logs.

Figure 5.9a describes the accuracy performance (i.e., F_1-score) of the anomaly detection methods while varying the proportion of synthetic log sequences injected into the dataset (i.e., injection ratio). The results show that sentence embedding methods, such as ALogSCAN and LogRobust, performed significantly better than DeepLog when injecting synthetic log sequences. For example, only at a 5% injection ratio, DeepLog reduced its F_1-score by 22%, whereas ALogSCAN and LogRobust suffered a decrease of 4% and 7%, respectively. This large reduction in DeepLog's accuracy is because its log key-based technique treats changed log events as entirely new templates, resulting in more false positives. In contrast, ALogSCAN and LogRobust's sentence embedding is more robust to log changes as it encodes log templates into multi-dimensional vectors.

Note both ALogSCAN and LogRobust experienced a decline in their accuracy performance as the injection ratio increased. However, ALogSCAN's accuracy decreased with a less steep, maintaining a higher F_1-score. For example, at a 30% injection ratio, ALogSCAN achieved an F_1-score above 90%, whereas LogRobust dropped to 84%. ALogSCAN's consistent accuracy performance is due to the use of BERT encoders that capture contextual information and encode variations in logs with similar vectors, making it more resilient to log changes.

For the second experiment, we simulated real-world conditions by introducing anomalies

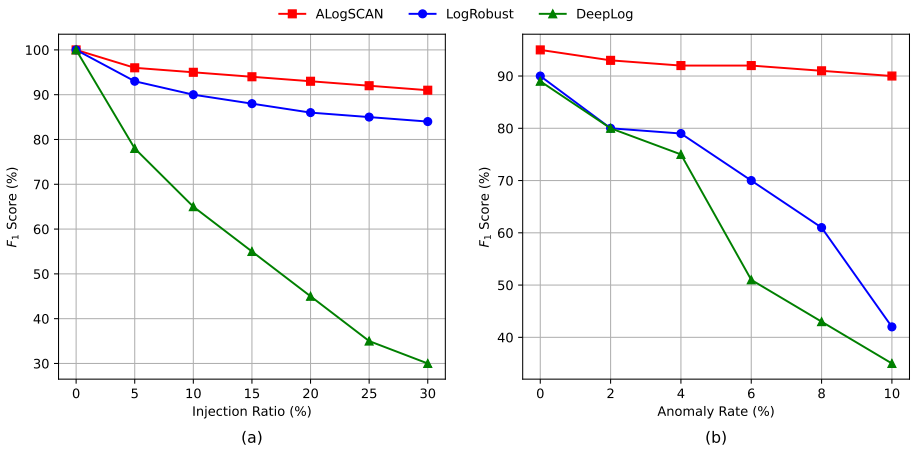


Figure 5.9: Accuracy performance of anomaly detection methods while varying (a) the proportion of injected synthetic log sequences and (b) the anomaly rate.

directly into the test subset of HDFS. This experiment evaluated the robustness of the anomaly detection methods while randomly injecting anomalies at different rates from 1% to 10%. Fig. 5.9b depicts the methods' accuracy performance (i.e., F_1-score) while varying the proportion of injected anomalies (i.e., anomaly rate). The results show the superior resilience of ALogSCAN as the number of anomalies increased. At an intermediate anomaly rate of 5%, the F_1-score of DeepLog and LogRobust dropped to 63% and 76%, respectively. Furthermore, as the anomaly rate increased to 10%, the accuracy of DeepLog and LogRobust further deteriorated up to producing an F_1-score around 40%. In contrast, ALogSCAN demonstrated robust accuracy performance by maintaining an F_1-score above 90% for all the range of the anomaly rate.

5.6.2.2 Ablation study

This section presents the ablation study of ALogSCAN, analyzing the contributions of its core components: network architecture, reconstruction techniques, and filtering. We conducted the experiments on three datasets: session-based HDFS and 100-log window BGL and ERDC.

(A) Network architecture and reconstruction techniques: Table 5.3 reports ALogSCAN's accuracy performance (i.e., F_1-score and MCC) using two network architectures (i.e., single-network and dual-network) and the three reconstruction paradigms: RCI, IRFI, and IRCI (see Section 5.5.4). The results show that ALogSCAN using the dual-network architecture consistently achieves higher accuracy than the single-network architecture across all datasets and reconstruction techniques. This behavior demonstrates that the self-supervised prediction task in the dual-network design enhances

Table 5.3: ALogSCAN's accuracy performance using different network types and reconstruction techniques.

Network	Reconstruction technique*	HDFS		100-log BGL		100-log ERDC	
		F_1-score	MCC	F_1-score	MCC	F_1-score	MCC
Single Network	RCI	78.58	78.80	91.52	91.26	79.49	78.01
	IRFI	71.43	71.22	89.39	90.95	88.46	87.69
	IRCI	92.13	91.53	95.96	95.34	89.21	89.30
Dual Network	RCI	92.62	92.71	91.66	91.84	88.36	89.13
	IRFI	75.97	76.88	92.81	94.18	88.86	88.12
	IRCI	95.83	95.59	94.83	94.83	89.57	89.54

* *Reconstruction with Complete Input (RCI), Infrequent-based Reconstruction with Frequent Input (IRFI), and Infrequent-based Reconstruction with Complete Input (IRCI).*

ALogSCAN's capacity to learn patterns from log sequences, leading to accurate anomaly detection. Regarding the reconstruction techniques, the results show that IRCI provides the best accuracy performance in the three datasets. Therefore, IRCI effectively captures more complex log patterns than the other reconstruction techniques. The importance of selecting the appropriate reconstruction technique is evident, as it directly affects ALogSCAN's ability to generalize and detect anomalies. We advocate for careful consideration when selecting reconstruction techniques to prevent potential information loss.

(B) Network architecture and filtering: Table 5.4 presents ALogSCAN's accuracy performance (i.e., F_1-score and MCC) using the two network architectures (i.e., single-network and dual-network) and with and without applying DFLF, the introduced filtering technique (see Section 5.5.3). The results show that DFLF significantly improves ALogSCAN's accuracy performance, providing higher and more stable accuracy values than without filtering across all datasets. This behavior is because DFLF enables the ML model to focus on rare log events, which is critical for effective anomaly detection. In contrast, when ALogSCAN skips filtering, the results highly varied due to the inclusion of noisy or irrelevant patterns.

5.6.2.3 Impact of configuration parameters

(A) Impact of reconstruction loss weights: The previous evaluation experiments used a reconstruction loss weight $\alpha = 100$ in the objective loss function L (see Eq. 5.4). We conducted a series of tests to understand better the impact of this hyperparameter in the accuracy of ALogSCAN by varying α from 10^{-3} to 10^4 in steps of 10x. As shown in Fig. 5.10a, ALogSCAN's F_1-score increased from 82% to 94% as α incremented, with a fluctuation between 10^{-2} to 10^{-1} and a minimum accuracy reduction for $\alpha > 100$. These results suggest that giving high importance to the reconstruction enables ALogSCAN

Table 5.4: ALogSCAN's accuracy performance using different network types, with and without filtering.

		HDFS		BGL		ERDC	
Network	**Filtering***	F_1-score	MCC	F_1-score	MCC	F_1-score	MCC
Single Network	No Filtering	38.95	38.72	82.26	83.57	33.40	33.35
	DFLF	90.67	91.01	92.64	93.39	80.12	80.32
Dual Network	No Filtering	41.20	42.23	83.71	83.82	34.20	35.77
	DFLF	95.83	95.59	97.16	94.83	89.57	89.54

* *Dynamic Frequency-based Log Filtering (DFLF)*

(a) (b)

Figure 5.10: ALogSCAN's accuracy performance while varying (a) the reconstruction loss weight α and (b) the DFLF's filtering ratio.

to distinguish between normal and anomalous log sequences. However, focusing only on reconstruction can lead to ALogSCAN's performance degradation. Based on this empirical analysis, we set $\alpha = 100$ as the optimal setting in our experiments.

(B) Impact of filtering ratio: We examined the effect of the DFLF's filtering ratio on ALogSCAN's anomaly detection performance. This experiment evaluated two types of filtering ratios: fixed and random. The fixed filtering ratio ranged from 0.05 to 0.9, whereas the random filtering ratio sampled a value from an empirically defined set $K = \{0.05, 0.1, 0.15, 0.2, 0.3\}$. As depicted in Fig. 5.10b, the random filtering ratio generally enables ALogSCAN to achieve a higher F_1-score than the fixed filtering ratio. Therefore, the random filtering ratio provides high accuracy and flexibility in hyper-parameter tuning across different datasets, allowing ALogSCAN to handle diverse log event frequencies and adapt to different log structures. Consequently, our experiments adopted the random filtering ratio to improve generalization.

5.7 Conclusion

In this chapter, we presented ALogSCAN, a self-supervised approach for log anomaly detection. ALogSCAN integrates three key components: DFLF filtering, IRCI reconstruction, and a dual-network architecture supported by three self-supervised learning techniques. The DFLF filtering strategy prioritizes learning normal patterns from infrequent log templates. The IRCI reconstruction approach reconstructs rare events by leveraging all templates in a log sequence. The dual-network architecture, enhanced by reconstruction, one-class classification, and KD, effectively captures less biased, discriminative normal patterns from infrequent log templates. This combination enables ALogSCAN to distinguish anomalous log sequences from those containing rare but normal templates with higher accuracy.

Our extensive experiments on three datasets demonstrated the significant performance gains of ALogSCAN over baseline anomaly detection methods. Across all datasets, ALogSCAN achieves an accuracy improvement of up to 79% (i.e., MCC and F_1-score), outperforming existing methods like PLELog and LogRobust. In addition, ALogSCAN excels in time efficiency, reducing the analysis time by up to 15x compared to methods like LogAnomaly and LogBERT. AE+IF exhibited a faster analysis time than ALogSCAN in the HDFS dataset, though by only 2 ms per log sequence. However, AE+IF suffered from poor accuracy, limiting its applicability in real-world scenarios where the trade-off between accuracy and analysis time is critical. In contrast, ALogSCAN consistently maintained an analysis time of 3-4 ms per log sequence while achieving the best accuracy performance. Therefore, our proposal provides the optimal balance between high accuracy and short analysis latency, suitable for timely anomaly detection tasks.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Detecting anomalies in real-time cloud environments using ML provides a proactive approach to enhance system reliability and efficiency. However, building and maintaining accurate ML models in such dynamic and heterogeneous environments is challenging. Key issues include concept drift, the interconnected and distributed nature of cloud services, and the evolution of log data formats, all of which degrade the performance of ML models over time and necessitate robust real-time adaptation mechanisms.

This thesis addresses these challenges by proposing algorithms to adapt ML models for anomaly detection in real-time cloud environments and overcome limitations related to data evolution and distribution changes. The main contributions are summarized as follows:

In Chapter 3, we introduced a real-time concept drift adaptation algorithm to tackle the challenges of sequential metric data in dynamic real-time cloud environments. Our approach leverages sliding and adaptive window-based methods combined with a performance-driven mechanism to effectively detect and adapt to concept drift in real-time. By integrating multi-source prediction and optimizing the model with a Genetic Algorithm, our proposed method achieves superior accuracy and time efficiency while ensuring scalability and adaptability.

In Chapter 4, we addressed the challenges of analyzing distributed traces in RT cloud systems for anomaly detection. We proposed a federated learning algorithm designed for asynchronous, real-time model updates, accommodating the distributed nature of cloud services. This approach minimizes communication and computation overhead, enabling timely anomaly detection with improved detection accuracy. Experimental evaluations demonstrated significant improvements in detection timeliness, effectiveness, and resource efficiency.

In Chapter 5, we tackled the complexities of working with contextual log data, such as unstable log formats, class imbalance, and reliance on labeled data. We proposed a robust, adaptive solution that accommodates evolving log structures and prioritizes rare

patterns, enabling accurate and efficient anomaly detection real-time. Comprehensive evaluations highlighted the effectiveness of our approach in improving detection performance and reducing dependency on labeled datasets.

6.2 Future Work

While this thesis introduced significant advancements in ML-based anomaly detection for real-time clouds, several research directions remain open. Future work should focus on further improving detection efficiency, interpretability, and adaptability while ensuring low-latency anomaly detection and mitigation in real-time cloud infrastructures.

6.2.1 Interpretable and Actionable ML-Based Anomaly Detection

While current research on ML-based anomaly detection primarily focuses on improving detection accuracy, developing interpretable and actionable models is equally critical—especially in real-time cloud environments, where fast and informed decision-making is essential for minimizing service disruptions. Operators must not only detect anomalies but also understand their causes instantly to ensure rapid mitigation.

Recent studies have explored anomaly explanation techniques by identifying key features that contribute most to an anomaly. Model-agnostic approaches, such as feature attribution and attention-based methods, can help explain why a model detects an event as anomalous. However, these methods often lack deep insights into how ML models make decisions in evolving real-time cloud environments, which can reduce their practical effectiveness for real-time troubleshooting and automated mitigation.

Future research should focus on integrating real-time feature attribution and actionable knowledge discovery directly into ML models for anomaly detection. Specifically:

- **Real-Time Anomaly Explanation**: Develop ML models that instantly identify and highlight key features responsible for anomalies, ensuring that cloud operators can respond promptly.
- **Context-Aware Interpretability**: Design models that correlate anomalies across multiple cloud layers (logs, traces, metrics) in real time, enabling faster root-cause analysis.
- **Automated Impact Quantification**: Develop methods that assess how anomalies affect service reliability, performance, and user experience, allowing systems to prioritize mitigation actions dynamically.

By embedding interpretability directly into anomaly detection models, real-time cloud systems can achieve faster response times, reduced operational overhead, and improved service reliability.

6.2.2 Holistic Anomaly Detection: Multi-Source Data Fusion in Real-Time Clouds

A promising direction for ML-based anomaly detection in real-time cloud environments is the development of a unified system that integrates and analyzes multiple data types, including metrics, traces, and logs. Existing approaches often rely on a single data source, which may cause models to miss critical cross-layer correlations that provide a complete picture of cloud failures.

For example, an anomaly detected in system metrics (e.g., a sudden spike in CPU usage) may be linked to trace anomalies (e.g., delayed API response times) and further contextualized by logs indicating an error or misconfiguration. Without a holistic view, anomaly detection models may misinterpret symptoms as isolated issues rather than part of a larger system failure. Developing a real-time multi-source anomaly detection system requires advancements in:

- **Low-Latency Data Integration**: Designing architectures that synchronize logs, traces, and metrics in real time, allowing anomaly detection models to process events as they occur.
- **Multi-Modal Learning for Anomaly Correlation**: Implementing ML techniques that learn dynamic relationships between different data sources, enabling better detection of complex, multi-layer anomalies.
- **Adaptive Feature Fusion for Concept Drift Handling**: Since real-time cloud behaviors evolve, anomaly detection must continuously adjust to new data relationships without requiring frequent retraining.

By integrating multi-source real-time data, cloud anomaly detection systems can achieve higher accuracy, reduced false positives, and faster failure diagnosis.

6.2.3 Reinforcement Learning for Adaptive Anomaly Detection and Resource Optimization

Reinforcement Learning (RL) offers significant potential for adaptive anomaly detection and cloud resource optimization, particularly in real-time environments where workloads and failure patterns change dynamically. Unlike traditional ML models, RL learns optimal responses over time, making it well-suited for the complex, dynamic nature of real-time cloud infrastructures.

In real-time cloud systems, RL could continuously adapt to evolving workloads and new anomaly patterns by learning from real-time interactions with cloud components. This capability could address critical challenges, such as minimizing anomaly detection latency to meet real-time SLA requirements, ensuring continuous learning despite evolving cloud

workloads, and optimizing cloud resource allocation to balance performance and cost in dynamic cloud environments.

Despite its potential, RL for real-time cloud is underexplored due to challenges in scalability, execution latency, and training environments. Future research should focus on:

- **Fast RL-Based Anomaly Detection**: Designing lightweight RL models that can detect and respond to anomalies in real-time with minimal computational overhead.
- **Adaptive RL for Evolving Cloud Conditions**: Using meta-learning and transfer learning to allow RL agents to quickly adapt to new cloud environments without extensive retraining.
- **Simulation Environments for RL in Real-Time Clouds**: Developing high-fidelity real-time cloud simulators where RL models can learn under realistic cloud workload conditions.
- **Energy-Aware RL for Cloud Resource Optimization**: RL can be leveraged to balance system performance and cost by dynamically adjusting resource allocation based on anomaly severity and real-time cloud demands.

By advancing RL-based adaptive anomaly detection and resource optimization, cloud systems can become more resilient, efficient, and capable of handling dynamic operational challenges in real-time.

Biblography

[1] M. Raeiszadeh, A. Saleem, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "A deep learning approach for real-time application-level anomaly detection in IoT data streaming," in *Proc. IEEE Consumer Communications & Networking Conference (CCNC)*, pp. 449–454, 2023.

[2] M. Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "Real-time adaptive anomaly detection in industrial IoT environments," *IEEE Transactions on Network and Service Management*, vol. 21, no. 6, pp. 6839–6856, 2024.

[3] M. Raeiszadeh, A. Ebrahimzadeh, A. Saleem, R. H. Glitho, J. Eker, and R. A. Mini, "Real-time anomaly detection using distributed tracing in microservice cloud applications," in *Proc. IEEE Cloud Networking (CloudNet)*, pp. 36–44, 2023.

[4] M. Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. Mini, "Asynchronous real-time federated learning for anomaly detection in microservice cloud applications," *IEEE Transactions on Machine Learning in Communications and Networking (TMLCN)*, vol. 3, pp. 176–194, 2025.

[5] M. Raeiszadeh, F. Estrada-Solano, R. H. Glitho, J. Eker, and R. A. Mini, "A data-driven approach for adaptive real-time log parsing in cloud environments," in *Proc. IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, pp. 173–178, 2024.

[6] M. Raeiszadeh, F. Estrada-Solano, R. H. Glitho, J. Eker, and R. A. Mini, "ALogSCAN: A self-supervised dual network for adaptive and timely log anomaly detection in clouds," *IEEE Transactions on Machine Learning in Communications and Networking (TMLCN)*, 2025.

[7] M. Raeiszadeh, F. Estrada-Solano, and R. H. Glitho, "Machine learning for anomaly detection in cloud-native architecture: A survey," in *IEEE Communication Magazine*, 2025. Under review.

[8] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[9] S. Chouliaras and S. Sotiriadis, "Real-time anomaly detection of nosql systems based on resource usage monitoring," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 6042–6049, 2019.

[10] P. Mell, T. Grance, *et al.*, *The NIST definition of cloud computing*. Computer Security Division, Information Technology Laboratory, National, 2011.

[11] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.

[12] Ericsson, "What is real-time cloud?," 2020. Accessed: October 17, 2024.

[13] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proceedings of the IEEE*, 2024.

[14] N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, and D. Pei, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proc. ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 527–539, 2021.

[15] S. J. Russell and P. Norvig, "Artificial intelligence (a modern approach)," 2010.

[16] N. Afshan and R. K. Rout, "Machine learning techniques for iot data analytics," *Big Data Analytics for Internet of Things*, pp. 89–113, 2021.

[17] L. Deng, D. Yu, *et al.*, "Deep learning: methods and applications," *Foundations and trends® in signal processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[18] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[19] X. Liu, F. Zhang, Z. Hou, L. Mian, Z. Wang, J. Zhang, and J. Tang, "Self-supervised learning: Generative or contrastive," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 857–876, 2021.

[20] V. Kumar, "Parallel and distributed computing for cybersecurity," *IEEE Distributed Systems Online*, vol. 6, no. 10, 2005.

[21] W. Wu, L. He, W. Lin, Y. Su, Y. Cui, C. Maple, and S. A. Jarvis, "Developing an unsupervised real-time anomaly detection scheme for time series with multi-seasonality," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[22] Y. T. Quek, W. L. Woo, and L. Thillainathan, "IoT load classification and anomaly warning in elv dc picogrids using hierarchical extended k-nearest neighbors," *IEEE Internet of Things Journal*, vol. 7, no. 2, pp. 863–873, 2021.

[23] Q. Xie, G. Tao, C. Xie, and Z. Wen, "Abnormal data detection based on adaptive sliding window and weighted multiscale local outlier factor for machinery health monitoring," *IEEE Transactions on Industrial Electronics*, 2022.

[24] N. Paulauskas and A. Baskys, "Application of histogram-based outlier scores to detect computer network anomalies," *Electronics*, vol. 8, no. 11, p. 1251, 2019.

[25] W. Hao, T. Yang, and Q. Yang, "Hybrid statistical-machine learning for real-time anomaly detection in industrial cyber-physical systems," *IEEE Transactions on Automation Science and Engineering*, 2021.

[26] A. B. Nassif, M. A. Talib, Q. Nasir, and F. M. Dakalbab, "Machine learning for anomaly detection: A systematic review," *IEEE Access*, vol. 9, pp. 78658–78700, 2021.

[27] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," in *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1939–1947, 2015.

[28] 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. ACM Internet Measurement Conference*, pp. 211–224, 2015.

[29] C. Li, L. Guo, H. Gao, and Y. Li, "Similarity-measured isolation forest: Anomaly detection method for machine monitoring data," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–12, 2021.

[30] A. Putina and D. Rossi, "Online anomaly detection leveraging stream-based clustering and real-time telemetry," *IEEE Transactions on Network and Service Management*, vol. 18, pp. 839–854, 3 2021.

[31] M. Hu, Z. Ji, K. Yan, Y. Guo, X. Feng, J. Gong, X. Zhao, and L. Dong, "Detecting anomalies in time series data via a meta-feature based approach," *IEEE Access*, vol. 6, pp. 27760–27776, 5 2018.

[32] M. Munir, S. A. Siddiqui, A. Dengel, and S. Ahmed, "DeepAnT: A deep learning approach for unsupervised anomaly detection in time series," *IEEE Access*, vol. 7, pp. 1991–2005, 2019.

[33] P. Zhao, X. Chang, and M. Wang, "A novel multivariate time-series anomaly detection approach using an unsupervised deep neural network," *IEEE Access*, vol. 9, pp. 109025–109041, 2021.

[34] X. Zhou, Y. Hu, W. Liang, J. Ma, and Q. Jin, "Variational LSTM enhanced anomaly detection for industrial big data," *IEEE Transactions on Industrial Informatics*, vol. 17, pp. 3469–3477, 2021.

[35] D. Wu, Z. Jiang, X. Xie, X. Wei, W. Yu, and R. Li, "LSTM learning with bayesian and gaussian processing for anomaly detection in industrial IoT," *IEEE Transactions on Industrial Informatics*, vol. 16, pp. 5244–5253, 2020.

[36] S. Chouliaras and S. Sotiriadis, "Real-time anomaly detection of NoSQL systems based on resource usage monitoring," *IEEE Transactions on Industrial Informatics*, vol. 16, pp. 6042–6049, 2020.

[37] P. Malhotra, L. Vig, G. Shroff, P. Agarwal, *et al.*, "Long short term memory networks for anomaly detection in time series," in *Proc. European Symposium on Artificial Neural Networks (ESANN)*, vol. 89, pp. 89–94, 2015.

[38] Y. Wang, X. Du, Z. Lu, Q. Duan, and J. Wu, "Improved lstm-based time-series anomaly detection in rail transit operation environments," *IEEE Transactions on Industrial Informatics*, 2022.

[39] V. Mothukuri, P. Khare, R. M. Parizi, S. Pouriyeh, A. Dehghantanha, and G. Srivastava, "Federated-learning-based anomaly detection for IoT security attacks," *IEEE Internet of Things Journal*, vol. 9, no. 4, pp. 2545–2554, 2021.

[40] J. Dromard, G. Roudière, and P. Owezarski, "Online and scalable unsupervised network anomaly detection method," *IEEE Transactions on Network and Service Management*, vol. 14, pp. 34–47, 3 2017.

[41] E. J. Spinosa, A. P. de Leon F. de Carvalho, and J. Gama, "Cluster-based novel concept detection in data streams applied to intrusion detection in computer networks," in *Proc. ACM Symposium on Applied computing*, pp. 976–980, 2008.

[42] L. Yu and Z. Lan, "A scalable, non-parametric method for detecting performance anomaly in large scale computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 1902–1914, 2015.

[43] L. Wang, S. Chen, and Q. He, "Concept drift-based runtime reliability anomaly detection for edge services adaptation," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[44] Y. Yang, X. Yang, M. Heidari, M. A. Khan, G. Srivastava, M. Khosravi, and L. Qi, "Astream: Data-stream-driven scalable anomaly detection with accuracy guarantee in iiot environment," *IEEE Transactions on Network Science and Engineering*, 2022.

[45] S. Saurav, P. Malhotra, V. TV, N. Gugulothu, L. Vig, P. Agarwal, and G. Shroff, "Online anomaly detection with concept drift adaptation using recurrent neural networks," in *Proc. ACM india joint international conference on data science and management of data*, pp. 78–87, 2018.

[46] D. Ippoliti, C. Jiang, Z. Ding, and X. Zhou, "Online adaptive anomaly detection for augmented network flows," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 11, no. 3, pp. 1–28, 2016.

[47] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.

[48] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 241–250, 2019.

[49] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao, "Self-supervised anomaly detection from distributed traces," in *Proc. IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pp. 342–347, 2020.

[50] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, *et al.*, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58, 2020.

[51] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *Proc. IEEE International Conference on Cloud Computing (CLOUD)*, pp. 179–186, 2019.

[52] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proc. Springer International Conference on Service-Oriented Computing (ICSOC)*, pp. 3–20, 2018.

[53] M. Jin, A. Lv, Y. Zhu, Z. Wen, Y. Zhong, Z. Zhao, J. Wu, H. Li, H. He, and F. Chen, "An anomaly detection algorithm for microservice architecture based on robust principal component analysis," *IEEE Access*, vol. 8, pp. 226397–226408, 2020.

[54] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A.-R. Sadeghi, "DïoT: A federated self-learning anomaly detection system for IoT," in *Proc. IEEE International conference on distributed computing systems (ICDCS)*, pp. 756–767, 2019.

[55] R. A. Sater and A. B. Hamza, "A federated learning approach to anomaly detection in smart buildings," *ACM Transactions on Internet of Things*, vol. 2, no. 4, pp. 1–23, 2021.

[56] H. T. Truong, B. P. Ta, Q. A. Le, D. M. Nguyen, C. T. Le, H. X. Nguyen, H. T. Do, H. T. Nguyen, and K. P. Tran, "Light-weight federated learning-based anomaly detection for time-series data in industrial control systems," *Computers in Industry*, vol. 140, p. 103692, 2022.

[57] Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, and M. S. Hossain, "Deep anomaly detection for time-series data in industrial IoT: A communication-efficient on-device federated learning approach," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6348–6358, 2020.

[58] T. T. Huong, T. P. Bac, D. M. Long, T. D. Luong, N. M. Dan, B. D. Thang, K. P. Tran, *et al.*, "Detecting cyberattacks using anomaly detection in industrial control systems: A federated learning approach," *Computers in Industry*, vol. 132, p. 103509, 2021.

[59] L. Cui, Y. Qu, G. Xie, D. Zeng, R. Li, S. Shen, and S. Yu, "Security and privacy-enhanced federated learning for anomaly detection in IoT infrastructures," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 5, pp. 3492–3500, 2021.

[60] J. Pei, K. Zhong, M. A. Jan, and J. Li, "Personalized federated learning framework for network traffic anomaly detection," *Computer Networks*, vol. 209, p. 108906, 2022.

[61] Y. Guo, Y. Wu, Y. Zhu, B. Yang, and C. Han, "Anomaly detection using distributed log data: A lightweight federated learning approach," in *Proc. IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2021.

[62] B. Li, Y. Wu, J. Song, R. Lu, T. Li, and L. Zhao, "DeepFed: Federated deep learning for intrusion detection in industrial cyber–physical systems," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5615–5624, 2020.

[63] B. Li, S. Ma, R. Deng, K.-K. R. Choo, and J. Yang, "Federated anomaly detection on system logs for the internet of things: A customizable and communication-efficient approach," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1705–1716, 2022.

[64] S. Ma, J. Nie, J. Kang, L. Lyu, R. W. Liu, R. Zhao, Z. Liu, and D. Niyato, "Privacy-preserving anomaly detection in cloud manufacturing via federated transformer," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 12, pp. 8977–8987, 2022.

[65] X. Kong, W. Zhang, H. Wang, M. Hou, X. Chen, X. Yan, and S. K. Das, "Federated graph anomaly detection via contrastive self-supervised learning," *IEEE Transactions on Neural Networks and Learning Systems*, 2024. IEEE Xplore Early Access.

[66] J. Jithish, B. Alangot, N. Mahalingam, and K. S. Yeo, "Distributed anomaly detection in smart grids: A federated learning-based approach," *IEEE Access*, vol. 11, pp. 7157–7179, 2023.

[67] T. Wang, W. Zhang, J. Xu, and Z. Gu, "Workflow-aware automatic fault diagnosis for microservice-based applications with statistics," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2350–2363, 2020.

[68] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting anomalies in microservices with execution trace comparison," *Future Generation Computer Systems*, vol. 116, pp. 291–301, 2021.

[69] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proc. ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 807–817, 2019.

[70] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.," in *IJCAI*, vol. 19, pp. 4739–4745, 2019.

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

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

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

[74] 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 International Conference on Software Engineering (ICSE)*, pp. 1448–1460, 2021.

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

[76] M. C. Domenech, L. P. Rauta, M. D. Lopes, P. H. Da Silva, R. C. Da Silva, B. W. Mezger, and M. S. Wangham, "Providing a smart industrial environment with the web of things and cloud computing," in *Proc. IEEE International Conference on Services Computing (SCC)*, pp. 641–648, 2016.

[77] J. M. DeAlmeida, C. F. Pontes, L. A. DaSilva, C. B. Both, J. J. Gondim, C. G. Ralha, and M. A. Marotta, "Abnormal behavior detection based on traffic pattern categorization in mobile networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4213–4224, 2021.

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

[79] L. Yang and A. Shami, "Iot data analytics in dynamic environments: From an automated machine learning perspective," *Engineering Applications of Artificial Intelligence*, vol. 116, p. 105366, 2022.

[80] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[81] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with Python," in *Proc. 9th Python in Science Conference*, 2010.

[82] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[83] I. Ullah and Q. H. Mahmoud, "A scheme for generating a dataset for anomalous activity detection in IoT networks," in *Proc. Canadian Conference on Artificial Intelligence*, pp. 508–520, Springer, 2020.

[84] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, "WUSTL-IIOT-2021 dataset for IIoT cybersecurity research." Washington University in St. Louis, USA, Oct. 2021.

[85] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

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

[87] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.

[88] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5149–5152, 2012.

[89] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, *et al.*, "Transformers: State-of-the-art natural language processing," in *Proc. International Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.

[90] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and content," Technical Report, RFC 7231, 2014.

[91] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, *et al.*, "Graph attention networks," in *Proc. International Conference on Learning Representations (ICLR)*, pp. 10–48550, 2017.

[92] R. Kiryo, G. Niu, M. C. Du Plessis, and M. Sugiyama, "Positive-unlabeled learning with non-negative risk estimator," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[93] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. International Conference on Learning Representations (ICLR) Engineering*, 2015.

[94] A. Avritzer, D. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, and H. Schulz, "PPTAM: production and performance testing based application monitoring," in *Proc. ACM/SPEC International Conference on Performance Engineering*, pp. 39–40, 2019.

[95] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[96] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Machine learning and systems*, vol. 2, pp. 429–450, 2020.

[97] P. Notaro, S. Haeri, J. Cardoso, and M. Gerndt, "Logrule: Efficient structured log mining for root cause analysis," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4231–4243, 2023.

[98] Y. Sui, Y. Zhang, J. Sun, T. Xu, S. Zhang, Z. Li, Y. Sun, F. Guo, J. Shen, Y. Zhang, *et al.*, "Logkg: Log failure diagnosis through knowledge graph," *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3493–3507, 2023.

[99] R. Xiao, H. Chen, J. Lu, W. Li, and S. Jin, "Allinfolog: Robust diverse anomalies detection based on all log features," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2529–2543, 2022.

[100] J. Qi, Z. Luan, S. Huang, C. Fung, H. Yang, H. Li, D. Zhu, and D. Qian, "LogEncoder: Log-based contrastive representation learning for anomaly detection," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1378–1391, 2023.

[101] 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," 2023.

[102] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.