Data Labeling for Fault Detection in Cloud: A Test Suite-Based Active Learning Approach

Prateek Bagora

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

May 2023

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Prateek Bagora**

Entitled: **Data Labeling for Fault Detection in Cloud: A Test Suite-Based Active Learning Approach**

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

Master of Computer Science

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. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Juergen Rilling

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Supervisor
Dr. R. Glitho

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

May 24, 2023 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Data Labeling for Fault Detection in Cloud: A Test Suite-Based Active Learning Approach

Prateek Bagora

Cloud computing enables ubiquitous on-demand network access to a shared pool of configurable computing resources with minimal management efforts from the user. It has evolved as a key computing paradigm to enable a wide variety of applications such as e-commerce, social networks, high-performance computing, mission-critical applications, and Internet of Things (IoT). Ensuring the quality of service of applications deployed in inherently complex and fault-prone cloud environments is of utmost concern to service providers and end users. Machine learning-based fault management solutions enable proactive identification and mitigation of faults in cloud environments to attain the desired reliability, though they require labeled cloud metrics data for training and evaluation. Moreover, the high dynamicity in cloud environments brings forth emerging data distributions, which necessitate frequent labeling of cloud metrics data stemming from an evolving data distribution for model adaptation. In this thesis, we study the problem of data labeling for fault detection in cloud environments, paying close attention to the phenomenon of evolving cloud metric data distributions. More specifically, we propose a test suite-based active learning framework for automated labeling of cloud metrics data with the corresponding cloud system state while accounting for emerging fault patterns and data or concept drifts. We implemented our solution on a cloud testbed and introduced various emerging data distribution scenarios to evaluate the proposed framework's labeling efficacy over known and emerging data distributions. According to our evaluation results, the proposed framework achieves about 41% higher weighted F1-score and 34% higher average Area Under One-vs-Rest Receiver Operating Characteristic curves (OvR ROC AUC score) than a system without any adaptation for emerging data distributions.

Acknowledgments

I would like to express my heartfelt gratitude to Concordia University for providing me with the opportunity to pursue this research and for the resources and support extended throughout the course of my study. This work would not have been possible without the unwavering support and guidance of numerous individuals whom I would like to acknowledge.

First and foremost, I am deeply grateful to my supervisor, Dr. Roch Glitho, for his invaluable guidance, continuous support, and insightful feedback, which helped me to make this work possible. I am truly indebted to him for his patience and understanding. I am fortunate to have had the privilege of working under his supervision.

I would like to express my sincere gratitude to Dr. Yann-Gaël Guéhéneuc and Dr. Juergen Rilling for serving as the committee members for my thesis.

I am profoundly grateful to Dr. Amin Ebrahimzadeh for providing me with invaluable mentorship, continuous support, and insightful suggestions, which helped me a lot to write my thesis and paper.

I would like to thank my colleagues at Telecommunication Services Engineering Lab, who provided me with a stimulating research environment and valuable resources.

I am forever indebted to my parents for their unwavering support and countless sacrifices that enabled me to accomplish what little I have. I am deeply grateful to my sisters for always being there for me. There are no words that can express my gratitude and love for you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter begins with an overview of the key terminologies associated with our research. Subsequently, we discuss the motivation and problem statement for this work. Next, we present a summary of the contributions of this work. Finally, we describe the organization of the rest of this thesis.

1.1 Definitions

1.1.1 Cloud Computing

Cloud computing is a paradigm that enables ubiquitous on-demand network access to a shared pool of configurable computing resources - data storage, networks, servers, applications, services, and more - that can be provisioned and released rapidly with minimal management effort or service provider interaction [4]. It comprises three main service models, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS enables consumers to provision fundamental computing resources like storage, network, and processing; PaaS enables consumers to access software development environments for application development, deployment, and management; and SaaS allows consumers to access cloud applications over the internet.

1.1.2 Fault

We can define a fault as the inability of a system to perform the desired tasks due to some abnormal condition or defect in one or more of its components, which can manifest into an erroneous system state and eventually result in the failure of the system to deliver the specified services [5]. For instance, cloud systems may encounter infrastructure faults like processor, memory, or network hogging or overutilization.

1.1.3 Fault Detection

We can define fault detection as the process of identifying the presence of a fault in the system and its components that may lead to a deviation from the expected operational behavior of the system by monitoring the system behavior or actively testing the system. Additionally, it may involve diagnosing the root cause of the fault and locating the component where the fault originated. The primary objective of fault detection is to identify and mitigate faults as early as possible to prevent system failures.

1.1.4 Test

A test is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [6]. Specifically in our context, a test is a sequence of actions executed programmatically to verify that a system or component meets the specifications by producing the desired output or behavior when provided with a specific input.

1.1.5 Test Suite

A test suite is a collection of tests intended to verify that a system is in conformance with the desired behavior, for example, Tempest developed by OpenStack community [7]. It may comprise functional, end-to-end, and other types of tests based on the objective of the test suite. Functional tests verify that each system feature complies with the functional specifications by producing outputs consonant with end-user expectations. End-to-end tests validate system functionality and performance from beginning to end, integrating all system components, and simulating live end-user scenarios.

1.1.6 Machine Learning

Machine learning is a field of inquiry focused on imitating intelligent human behavior of learning from experiences in machines, predominantly by employing algorithms that can learn to perform the desired tasks from sample data, identifying relevant underlying patterns. Supervised machine learning algorithms, the most common form of machine learning, build mathematical models from a data set with the supervision of data labels associated with data points. On the other hand, unsupervised machine learning algorithms identify hidden patterns in a data set based on similarities or differences between data points without the supervision of data labels.

1.1.7 Active Learning

Active learning is a subfield of machine learning where a machine learning algorithm interactively queries an oracle, regarded as a source of ground truth, for labeling the most informative data points and progressively improves itself with the supervision of the labels obtained. While traditional passive learning systems induce a hypothesis to explain the available labeled training data, active learning systems eagerly develop and test new hypotheses through a continuative and interactive learning process [8].

1.1.8 Transfer Learning

Transfer learning is a branch of machine learning that focuses on applying the knowledge acquired while solving one task to effectively and efficiently learn to solve a new task by leveraging the similarity of data between the two tasks [9]. More formally [10], given a source domain and task $\{D_S, T_S\}$ and a target domain and task $\{D_T, T_T\}$, transfer learning tries to use knowledge from the source to improve the learning of the target task, where $D_S \neq D_T$ or $T_S \neq T_T$, where a domain is defined by the input space X and the marginal probability distribution of the inputs $P(X)$ and a task is defined by the output space Y and the posterior distribution of the output $P(Y|X)$.

1.1.9 Data Labeling

Data labeling entails the assignment of one or more meaningful and informative labels to data points, which provide the context of learning to a machine learning algorithm. These labels guide the machine learning algorithm to find patterns in the data set that map input attributes to desired outputs by providing supervisory ground truth while training. Moreover, these labels enable the performance evaluation of trained machine learning models against the ground truth. While common human wisdom may suffice for labeling tasks like annotating generic images with objects in the frame, domain expertise is essential for labeling tasks like annotating X-ray images with malignant tumors.

1.2 Motivation and Problem Statement

Cloud environments are inherently complex and prone to faults due to their dynamic, heterogeneous, and distributed nature. If not detected and mitigated in time, faults can accumulate and propagate, leading to severe performance degradation and downtime. Therefore, ensuring the quality of service of applications deployed in cloud environments always concerns service providers and end users. One of the most accurate techniques for fault detection in cloud environments is periodically executing a test suite on a live cloud system. However, frequent execution of a test suite is expensive with regard to computational time and cloud resource consumption.

Machine learning techniques have recently received the utmost attention for fault detection in cloud environments. Supervised machine learning solutions [11] require labeled data for training. Though unsupervised machine learning solutions [12] do not require labeled data for training, they still need it for evaluation. Furthermore, their applicability and performance are often inferior to those of supervised machine learning solutions. While acquiring labeled data is necessary, data labeling is a significant bottleneck in machine learning and has not been studied much for the purpose of fault detection in cloud environments.

Most machine learning solutions for fault detection in cloud environments acquire the labeled metrics data by fault injection mechanism. This mechanism entails a manual injection of faults into the system followed by assigning the metrics collected in the fault injection time frame with the injected faults as labels. However, such solutions rely on the assumption that the metric data distributions produced by the fault injection adequately characterize the statistical nature of real-world fault scenarios. Moreover, the dynamicity of cloud environments is bound to change the statistical nature of metric data distributions, introducing emerging fault patterns and data or concept drifts, which necessitates retraining on the newly acquired data. However, frequent data acquirement with the fault injection mechanism is practically infeasible, as the fault injections may disrupt the cloud production workloads.

There exist several data labeling methods in the literature. Crowd-based techniques like active learning [13] and crowdsourcing [14] delegate the labeling task to the crowd for manual labeling. However, labeling cloud metrics data manually is challenging even for domain experts. Self-labeled [15] and graph-based label propagation [16] techniques exploit the existing labeled data to infer the labels for the remaining unlabeled data based on the premise that the unlabeled data follow the same distribution as the labeled data, which does not hold for emerging cloud metric data distributions. The weak supervision [17] approach of data programming involves a programmatic generation of data labels using heuristic labeling functions in large quantities to suffice for inaccuracies in these labels. However, weakly supervised learning has comparatively inferior generalization and accuracy.

As existing approaches are unsuitable for data labeling for fault detection in cloud environments, we need a mechanism for automatically labeling cloud metrics data with the corresponding cloud system state, which may be a normal system state or a system state with one or more faults, while accounts for emerging cloud metric data distributions.

1.3 Thesis Contributions

The primary contributions of this thesis can be summarized as follows:

- We propose a test suite-based active learning framework for automatically labeling cloud metrics data with multiple class labels corresponding to various cloud system states.
- Our proposed data labeling framework leverages transfer learning to adapt to unknown metric data distributions comprising emerging fault patterns and data or concept drifts.

- We evaluate the labeling performance of the proposed framework via experiments carried out on a cloud testbed with various known and unknown metric data distribution scenarios.

1.4 Thesis Organization

The remainder of this thesis is structured as follows. We discuss the key concepts related to this research work in Chapter 2. In Chapter 3, we present an illustrative scenario to explain the data labeling problem for fault detection in cloud environments to derive the requirements. Subsequently, we evaluate the existing literature against the derived requirements. We present the proposed framework for labeling cloud metrics data in Chapter 4 with a detailed explanation of its components. We discuss our experimental cloud testbed setup and framework prototype implementation details, like parameter settings and programming tools used, in Chapter 5. Subsequently, we present our evaluation results. Chapter 6 concludes this thesis by summarizing our contributions and identifying future research directions.

Chapter 2

Background Information

This chapter presents the background concepts relevant to the research domain of this thesis. In the first section of this chapter, we define cloud computing and discuss its essential characteristics. Subsequently, we describe the layers of cloud computing and the major cloud deployment models. We then discuss virtualization, a key enabling technology of cloud computing. In the second section of this chapter, we define faults and discuss the common types of faults that may occur in the cloud. In the third section of this chapter, we define fault detection and discuss some well-known techniques for fault detection in the cloud. In the fourth section of this chapter, we define test suites and discuss a few types of tests that comprise test suites. In the fifth section of this chapter, we define machine learning, present a taxonomy of machine learning techniques, and discuss deep learning. In the sixth section of this chapter, we define active learning and present a taxonomy of active learning techniques. In the seventh section of this chapter, we define transfer learning and present a taxonomy of transfer learning techniques. In the eighth section of this chapter, we define data labeling and present a taxonomy of data labeling techniques.

2.1 Cloud Computing

In this section, we first define cloud computing. Then, we discuss the essential characteristics of cloud computing, the layers of cloud computing, and various cloud deployment models. Finally, we talk about virtualization, a key enabling technology for cloud computing.

2.1.1 Definition

Emerging as one of the most powerful technologies of the modern era, cloud computing is the culmination of various existing technologies like virtualization, web services, utility computing, distributed computing, and others put together into a modern operating model. Hence, we can find many definitions of cloud computing focusing on different aspects of

the technology. In [4], they gather various available definitions and integrate them into an all-encompassing definition of cloud computing, "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs". Also, the National Institute of Standards and Technology (NIST) defines cloud computing as a "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [18].

2.1.2 Essential Characteristics

In this section, we briefly discuss the characteristics of cloud computing that make it a distinct paradigm [18] [19].

- **On-Demand Self-Service** The consumers can unilaterally provision computing capabilities at any time without human interaction with the service provider.
- **Multi Tenancy** Multiple consumers with disparate requirements share the same service provider resources in an isolated manner which facilitates optimum resource utilization and cost.
- **Broad Network Access** The consumers can access the capabilities over the network through standard mechanisms using heterogeneous thin or thick client platforms like workstations and mobile phones.
- **Measured Service** The consumers are charged only for the resources they use by leveraging a metering capability based on the type of resource used, the number of hours of usage, and other criteria.
- **Scalability** The resources offered can expand vertically within the existing infrastructure or horizontally with additional infrastructure to sustain increased workloads and shrink back when the workload decreases to save costs.
- **Elasticity** The system adapts to workload changes by provisioning and de-provisioning resources automatically and dynamically, such that at each point in time, the available resources match the current demand as closely as possible.

2.1.3 Cloud Computing Layers

Based on the abstraction level of the capability offered and the service model of the service provider, cloud computing services can be organized into three layers, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [18] [19].

Infrastructure as a Service

IaaS is the lowest layer of the cloud. Here, the capability offered to the consumer is to provision fundamental computing resources like processing, storage, and networks from the provider, where the consumer can deploy and run arbitrary software, including operating systems and applications. Some well-known examples of IaaS are Amazon Elastic Compute Cloud, Microsoft Azure, Google Compute Engine, and IBM Cloud.

Platform as a Service

PaaS is the middle layer of the cloud. Here, the capability offered to the consumer is an environment where developers can create and deploy applications using programming languages, libraries, services, and tools supported by the provider, without dealing with the underlying cloud infrastructure, including network, servers, operating systems, or storage. Some well-known examples of PaaS are Microsoft Azure, Google App Engine, and Cloud Foundry.

Software as a Service

SaaS is the highest layer of the cloud. Here, the capability provided to the consumer is to use the provider's cloud-hosted applications through web portals or program interfaces. For instance, traditional desktop applications like word processing are now accessible as SaaS over the web, alleviating the burden of software maintenance for consumers and simplifying development and testing for providers. Some well-known examples of SaaS are Google Docs, Microsoft Office 365, Salesforce, and Dropbox.

2.1.4 Cloud Deployment Models

Depending on the owner(s) and user(s), the responsibility of management and desired security, or the physical location and distribution of the underlying infrastructure, we can have different cloud deployment models [18] [19]. We describe three major cloud deployment models in brief in this section.

Public Cloud

A public cloud, generally owned by a big corporation with the infrastructure located on its premises, is available to the general public on a pay-per-use basis. While it does not require any initial capital investment in the infrastructure from the users, it provides limited control over data, network, and security settings.

Private Cloud

A private cloud is used exclusively by a single organization and is built and managed by an external provider, the organization itself, or some combination of the two. While it requires a high initial investment, it provides the highest degree of control over security, performance, reliability, and other factors.

Hybrid Cloud

A hybrid cloud is a combination of public and private clouds consolidating the best of both worlds. For instance, a good split between a private and a public cloud would enable an organization to run mission-critical applications with sensitive data in an on-premise private cloud while having a connected public cloud to facilitate on-demand resource expansion and contraction.

2.1.5 Virtualization

Virtualization enables the creation of virtual instances of physical resources like servers, networks, or storage systems through abstraction or emulation. Thus, it allows the decoupling of hardware infrastructure and software programs running on it to facilitate efficient resource utilization and multi-tenancy. Virtualization is one of the key enabling technologies in the realization of cloud computing through virtual machines or containers.

Virtual Machine

A virtual machine is a software-defined machine that provides the same behavior as a real machine, producing the same inputs and outputs. A hypervisor manages the virtual machines, enabling operations like creation, deletion, migration, and resource configuration of virtual machines and ensuring isolation between them. However, each virtual machine includes a full-blown operating system in addition to the desired application, which may not be the most efficient way to utilize the resources.

Container

A container comprises a single executable software package that bundles application code together with the runtime environment. A container runtime engine hosts the containers, which run as processes on the same operating system kernel with isolated file systems and computing resources. Moreover, a container orchestration platform automates the deployment, management, scaling, and networking of the containers. By sharing the operating system kernel, containers utilize the resources more efficiently than virtual machines.

2.2 Fault

In this section, we first define and briefly discuss fault. Subsequently, we provide a brief overview of various types of faults.

2.2.1 Definition

We can define a fault as the inability of a system to perform the desired tasks due to some abnormal condition or defect in one or more of its components [5][20][21]. Faults often lie latent in system components and manifest into errors only when they become active over time. We can define an error as an incorrect internal system state where the observed value of some quantity produced by the system deviates from its expected value due to the manifestation of some fault. If an error is not handled or hidden from the rest of the system, it can propagate to other system components and lead to a failure as it reaches the system's service interface. We can define a failure as an event where the system fails to deliver the specified services or functionalities to users or other systems.

For better understanding, let us consider a system with a memory leak in one of its software components due to a bug in a piece of its source code. Until the execution of this code, this fault remains dormant in the system. However, when this code executes, it does not free the allocated memory as it ideally should and manifests itself into an erroneous system state. As long as the amount of this unnecessarily allocated memory is sufficiently small, this error will neither be detected nor prevent the system from delivering the specified services. Nevertheless, the available memory resources will deplete over time with multiple executions of this code until no memory is left for some allocation and the error is detected. If the error is handled, for instance, by completing the desired operation on a redundant unit, this error will not lead to a failure. But if the system fails to perform some necessary operation and deliver the specified services due to this unsuccessful memory allocation, it will lead to a system failure.

While this distinction among faults, errors, and failures is theoretically precise, it is seldom applied in practice. Like the majority of the literature, we use the terms faults, errors, and failures interchangeably in this thesis.

2.2.2 Types of Faults

There are numerous ways to classify faults based on context and frame of reference [20] [21][22]. We discuss a few of these classifications in this section.

- Based on the computing resource the fault lies in, we can classify the faults as **processor**, **memory**, **network**, and **storage** faults. In its simplest form, a processor crash is a processor fault. In a virtualized environment, we may encounter a processor fault when the processes on a virtual machine with excessive workload face a dearth of CPU cycles.

Also, we may encounter a CPU fault when the Linux kernel throttles a container of further CPU cycles when its CPU consumption exceeds its set CPU limit. In its simplest form, a memory crash is a memory fault. In a virtualized environment, we may encounter a memory fault when a virtual machine under excessive workload runs out of memory to allocate to processes. Also, we may encounter a memory fault when the Linux kernel OOM (Out of Memory) kills a container when its memory consumption exceeds its set memory limit. Network congestion issues and loss of packets on a network qualify as network faults. We may encounter storage faults when a hard disk drive corrupts or runs out of storage space.

- Based on the nature of the occurrence, we can classify the faults as **transient**, **intermittent**, and **permanent** faults. A transient fault occurs once and then disappears, for instance, a momentary loss of network connectivity to a system component. An intermittent fault reoccurs occasionally, for instance, as a result of varying software states based on load or activity. A permanent fault occurs and persists until it has been addressed, for instance, a hardware component crash.
- Based on the nature of the output, we can classify the faults as **fail-silent** and **byzantine** faults. In a fail-silent fault scenario, the fault-affected component stops working, producing either no result or a result indicating component failure. In a byzantine fault scenario, the fault-affected component produces arbitrary erroneous outputs, leading to unpredictable conditions within the system. In other words, a single byzantine fault within a component could result in the component responding with various different errors.
- Based on how other system components perceive them, we can classify the faults as **response**, **timing**, **omission**, and **crash** faults. If a component produces an incorrect response, we call it a response fault. If a component delivers the correct response but later than the expected time interval, we call it a timing fault. If a component is unable to send or receive responses to or from other components, we call it an omission fault. If a component suffers from an omission fault and then continues to be completely unresponsive, we call it a crash fault.

2.3 Fault Detection

In this section, we first define fault detection. Subsequently, we briefly discuss some well-known techniques for fault detection in cloud environments.

2.3.1 Definition

We can define fault detection as the process of identifying the presence of a fault in the system and its components that may lead to a deviation from the expected operational behavior of the system by monitoring the system behavior or actively testing the system.

Additionally, it may involve diagnosing the root cause of the fault and locating the component where the fault originated. The primary objective of fault detection is to identify and mitigate faults as early as possible to prevent system failures.

2.3.2 Fault Detection Techniques

In the following, we discuss some well-known techniques for fault detection in cloud computing domain.

- **Performance Metrics Monitoring:** Performance metrics monitoring involves monitoring various performance metrics and system variables, like processor usage, memory usage, disk usage, swap usage, network traffic, average requests, average response times, and service uptime, to detect abnormal patterns or deviations from expected values, using tools like Prometheus Alerting [23]. Essentially, we set up upper and lower bounds for each metric based on prior knowledge of performance constraints and historical data analysis. Whenever any observed metric violates a threshold, a fault alarm is triggered.

- **System Log Mining:** System logs are records of system states and significant events at various critical points, which can provide valuable information about the health and performance of the system. System log mining employs various data mining techniques for automatically exploring and analyzing large volumes of log data to glean meaningful patterns and informative trends and exploit them to identify abnormalities and potential faults in the system [24].

- **Test Suite:** The test suite approach involves periodically executing a series of tests designed to verify that the system and its components are available and produce results in conformance with the desired behavior [25]. A test suite may comprise a wide variety of tests, for instance, functional tests to verify the availability and correctness of services offered and performance tests to ensure that the services meet the response time specifications.

- **Machine Learning:** Machine learning algorithms can automatically identify and learn the correlations between system performance metrics and faults to facilitate fault detection. Supervised machine learning solutions learn to map the abnormalities in performance metrics to corresponding faults by learning from example pairs of performance metric vectors and supervisory fault labels [11]. Unsupervised machine learning solutions learn the probability distributions of performance metrics corresponding to a fault-free system state. Subsequently, they identify the performance metric instances significantly deviating from these probability distributions as possible faults [12].

2.4 Test Suite

In this section, we first define and briefly discuss test suite. Subsequently, we provide a brief overview of various types of tests that may comprise a test suite.

2.4.1 Definition

A test suite is a collection of tests intended to verify that a system is in conformance with the desired behavior. A test is a sequence of actions performed to validate a specific functionality of a system. For instance, a test suite for an e-commerce application might consist of tests corresponding to functionalities like browsing products, adding products to the cart, viewing the cart, checkout, and sending a confirmation email. A test has a specified expected output to match against the produced output to deduce if it fails or succeeds. A test might also have a specified pre-condition that should be satisfied before running it, for instance, to run the test corresponding to checkout, the cart should not be empty. In addition to logically grouping the tests, a test suite may specify a sequence of execution for these tests based on their interdependencies. For instance, if the test corresponding to adding products to the cart fails, there is no need to execute the test corresponding to checkout.

2.4.2 Types of Tests

In this section, we discuss some types of tests that generally comprise a test suite [26][27].

- **Functional Tests:** Functional tests verify that a system under test meets the functional requirements as expected by users, checking the final output while ignoring the intermediate states of the system.
- **End-to-End Integration Tests:** End-to-End Integration tests verify that the different components of a system work properly together, focusing on the interface, communication, and data flow among them, mimicking system workflows from beginning to end as expected in real-world scenarios.
- **Regression Tests:** Regression tests verify that the unchanged functionalities of the system are not affected by the modifications made to the system for adding, deleting, or updating system functionalities or fixing bugs.
- **Performance Tests:** Performance tests examine the system speed, stability, reliability, scalability, and resource usage under various workloads, for instance, to ensure acceptable response times or error rates from the system.
- **Smoke Tests:** Smoke tests verify the basic and critical functionalities of the system under test at a high level for quick validation and to decide whether or not we can run more expensive tests.

2.5 Machine Learning

In this section, we first define machine learning. Subsequently, we briefly discuss a taxonomy of machine learning techniques. Finally, we talk about deep learning, a branch of machine learning.

2.5.1 Definition

Machine learning is a field of inquiry focused on imitating intelligent human behavior of learning from experiences in machines, predominantly by employing algorithms that can learn to perform the desired tasks themselves from representative sample data, identifying relevant underlying patterns with no explicit programming. This branch of artificial intelligence mainly focuses on designing algorithms that can assess provided data and learn by themselves without any human intervention, not necessarily by gaining knowledge consciously like humans but by identifying the patterns present in the provided data [28].

2.5.2 Taxonomy of Machine Learning Techniques

There are four main classes of machine learning techniques based on the learning procedure, supervised, unsupervised, semi-supervised, and reinforcement learning [28][29].

Supervised Learning

Supervised machine learning algorithms involve training over a data set consisting of pairs of data instances and supervisory data labels to build mathematical models that can map unlabeled data instances with respective data labels. We can distinguish supervised learning problems into classification problems where the data labels have discrete values and regression problems where the data labels have continuous values.

Unsupervised Learning

Unsupervised machine learning algorithms identify hidden patterns in a data set based on similarities or differences between data points without the supervision of data labels. The most common form of unsupervised learning is clustering, which aims to organize the data points into similarity groups called clusters, such that the data points in the same cluster are similar to each other and dissimilar to the data points in other clusters.

Semi-Supervised Learning

Semi-supervised machine learning algorithms, conceptually situated between supervised and unsupervised learning, aim to use both labeled and unlabeled data for learning, to

harness the large amounts of unlabeled data available in many use cases in combination with typically smaller sets of labeled data to attain improved efficacy.

Reinforcement Learning

Reinforcement learning techniques enable an agent to learn in an interactive environment by trial and error using feedback from its actions and experiences. The agent seeks to maximize cumulative reward while receiving rewards for desired behaviors and penalties for undesired behaviors to achieve an optimal solution. Unlike unsupervised learning, there is a form of supervision in reinforcement learning. However, this supervision is not in the form of a desired output for each data instance, like supervised learning.

2.5.3 Deep Learning

Conventional machine learning techniques require considerable domain expertise and careful feature engineering to transform the raw data into a feature vector with a data representation suitable for the learning algorithm. This manually engineered feature representation is essential for a performant application of a conventional machine learning algorithm. Deep learning is a branch of machine learning based on artificial neural networks and representation learning that addresses these shortcomings of conventional machine learning.

Representation learning techniques allow a system to automatically discover the representations suitable to perform a task from the raw data and learn to perform the corresponding task using them [30]. An artificial neural network is a computational model inspired by the biological neural networks that constitute the brain. It is a massively parallel distributed processor composed of simple processing units called neurons, which can acquire experiential knowledge and make it available for later use [31]. It stores the acquired knowledge as interneuron connection strengths called weights. Moreover, it acquires this knowledge from its environment through a learning process, adjusting these weights.

Deep learning techniques are representation learning techniques with multiple levels of representation, obtained by the composition of multiple layers of simple but non-linear neuron-based modules, where each of these layers transforms the representation at one level, starting from the raw input, into a representation at a slightly higher abstraction level [30]. With the composition of enough such transformations, deep neural networks can learn very complex functions, discovering intricate structures in high-dimensional data. Furthermore, these layers of feature representations are not designed by human engineers but learned from data using a general-purpose learning procedure, unlike conventional machine learning. Deep learning techniques can be applied to both supervised and unsupervised learning tasks.

2.6 Active Learning

In this section, we first define and briefly discuss active learning. Subsequently, we go through a taxonomy of active learning techniques.

2.6.1 Definition

Active learning is a subfield of machine learning where a machine learning algorithm or a committee of machine learning algorithms interactively query an oracle, regarded as a source of ground truth, for labeling the most informative data points, and progressively improve their accuracy with the supervision of the labels obtained. While traditional passive learning systems induce a hypothesis to explain the available labeled training data, active learning systems eagerly develop and test new hypotheses through a continuative and interactive learning process [8]. Essentially, an active learning system comprises two primary components:

- **Query System:** The query system poses queries to the oracle for labeling the most informative data points selected using a querying strategy. The key challenge in active learning systems is to design precise strategies for querying based on the objective of active learning.
- **Oracle:** The oracle responds to the query by annotating the delegated data points with labels, which act as the ground truth for learning. Although it is conventionally an expert human labeler, it can comprise some other cost-driven data acquisition system or any other methodology.

2.6.2 Taxonomy of Active Learning Techniques

We can classify active learning techniques based on querying strategies into the following categories [32].

- **Heterogeneity-Based Models:** Heterogeneity-based models attempt to select data points for queries that are most heterogeneous or dissimilar to the data previously seen by the learners. In classification problems, the regions around the decision boundaries are often occupied by data points from different classes that may help the learner to map the contours of separation between different classes using a small number of examples. Heterogeneity-based models attempt to select such data points by exploiting their characteristics like high class label uncertainties in learners' predictions and disagreements between different learners in the committee.
  - **Uncertainty Sampling:** In uncertainty sampling, the learner selects the data points it is least certain how to label for querying to the oracle. A simple example of uncertainty

sampling is a binary classification problem where a probabilistic classifier selects the next data instances for querying where the predicted probability of the most probable label is close to 0.5. Another approach to uncertainty sampling is margin sampling, where a probabilistic classifier selects the next data instances for querying where the difference between the predicted probabilities of the first and second most probable labels is relatively smaller.

  - **Query-by-Committee:** In query-by-committee, we employ a committee of classifiers trained on the available labeled data set to make class label predictions for the unlabeled data points. Here, we consider the data points where the classifiers disagree the most as informative and delegate them to the oracle for labeling. The committee of classifiers can be constructed by varying the model parameters of a particular classifier through sampling or using a bag of different classifiers.
  - **Expected Model Change:** The expected model change approach selects the data points that will result in the most change in the current model with the intuition that such data points differ from what the current model already knows. Essentially, it selects the data points that correspond to the highest change in the gradient of the training objective function with respect to the current model parameters.

- **Performance-Based Models:** Performance-based models attempt to select data points for queries that would optimize the learner's performance by reducing the error or variance. While heterogeneity-based models may select data points from noisy, unrepresentative, or outlier regions of the feature space leading to degradation of the learner's performance, performance-based models overcome this shortcoming by directly considering the effect of adding the queried data points on the learner's performance for remaining unlabeled data points.

  - **Expected Error Reduction:** In expected error reduction approach, we estimate the resulting error in the learner's predictions on the remaining unlabeled data set if we provide it with the label of some new data point from the unlabeled data set. Subsequently, we select the data point for querying that minimizes the expected future error. However, this approach may be the most prohibitively expensive approach as it requires re-training a new model for each possible label of each candidate data point from the sample of unlabeled data points to estimate the expected future error reduction.
  - **Expected Variance Reduction:** As noted in [33], we can express the overall generalization error as a sum of the true label noise, model bias, and variance. Among these, the variance is highly dependent on the data points selected. Thus, we can typically reduce the error by reducing the variance. The expected variance reduction approach selects the data points for querying that minimize the learner's future error by minimizing the variance, which can be expressed in closed form, to achieve greater computational efficiency over the expected error reduction approach.

- **Representativeness-Based Models:** Representativeness-based models attempt to select data points for queries that better resemble the overall input data distribution. We can

achieve this by weighting dense regions of the input space to a higher degree during the querying process. Hence, this method combines the heterogeneity behavior of the data points with a representativeness function to select data points for querying that are informative yet not outliers.

2.7 Transfer Learning

In this section, we first define and briefly discuss transfer learning. Subsequently, we go through a taxonomy of transfer learning techniques.

2.7.1 Definition

Transfer learning is a branch of machine learning that focuses on applying the knowledge acquired while solving one task to effectively and efficiently learn to solve a new task, especially beneficial when the amount of labeled data available for the new task is limited [9].

According to [10], let a domain D be defined by a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$, where $X = \{x_1, ..., x_n\} \in \mathcal{X}$, denoted by $D = \{\mathcal{X}, P(X)\}$. For a domain D, let a task T be defined by a label space $\mathcal{Y}$ and an objective predictive function $f(\cdot)$, denoted by $T = \{\mathcal{Y}, f(\cdot)\}$. The function $f(\cdot)$ is learned from a training data set comprising pairs of $\{x_i, y_i\}$, where $x_i \in X$ and $y_i \in \mathcal{Y}$, and can be used to predict the corresponding label, $f(x)$, of a new data instance x. Then, given a source domain and task $\{D_S, T_S\}$ and a target domain and task $\{D_T, T_T\}$, transfer learning tries to use knowledge from the source to improve the learning of the target task, where $D_S \neq D_T$ or $T_S \neq T_T$.

2.7.2 Taxonomy of Transfer Learning Techniques

We can classify transfer learning techniques based on various settings among source and target domains and tasks into the following categories [10].

- **Inductive Transfer Learning:** Inductive transfer learning aims to improve the learning of the target predictive function $f_t(\cdot)$ in D_T using the knowledge in D_S and D_T, where $T_S \neq T_T$. Here, the target task is different from the source task regardless if the source and target domains are the same or not. It requires the availability of some labeled training data instances in the target domain to induce the target predictive function.
- **Transductive Transfer Learning:** Transductive transfer learning aims to improve the learning of the target predictive function $f_t(\cdot)$ in D_T using the knowledge in D_S and D_T, where $D_S \neq D_T$ and $T_S = T_T$. Here, the source and target tasks are the same, while the source and target domains are different. It requires the availability of some unlabeled training data instances in the target domain to obtain the marginal probability for the target data.

- **Unsupervised Transfer Learning:** Unsupervised transfer learning aims to improve the learning of the target predictive function $f_t(\cdot)$ in D_T using the knowledge in D_S and D_T, where $T_S \neq T_T$ and Y_S and Y_T are latent variables, such as clusters or reduced dimensions and hence not observable. Here, we do not have any labeled training data in the source and target domains.

We can classify transfer learning techniques based on what part of the knowledge we transfer from source to target domains or tasks into the following categories [10].

- **Instance Transfer Approach:** The instance transfer approach focuses on reusing certain parts of the data in the source domain for learning in the target domain through techniques like instance reweighting. Instance reweighting involves adjusting the weights of the source domain instances by assigning higher weights to the instances that are more similar to the target domain to facilitate learning from the most relevant source domain instances, as in [34].

- **Feature Representation Transfer Approach:** The feature representation transfer approach focuses on exploiting the source domain to learn a good feature representation for the target domain by minimizing the divergence between the source and target domains. A simple example is using the high-level feature representation extracted by a pre-trained deep learning model trained on a large amount of labeled data from a related source domain as input to a model trained on a limited amount of labeled data from the target domain.

- **Parameter Transfer Approach:** The parameter transfer approach assumes that individual models for related source and target tasks should share some parameters or prior distributions of the hyperparameters. Thus, it focuses on discovering and exploiting these shared parameters or priors for knowledge transfer across the tasks. A simple example is employing a pre-trained deep learning model with its learned parameters from the source domain data as the starting point, adapting it to the target domain or task by fine-tuning its parameters on the target domain data using a smaller learning rate.

- **Relational Knowledge Transfer Approach:** The relational knowledge transfer approach deals with relational domains, where the data are not independent and identically distributed and can be represented by multiple relations among them, such as social network data. Assuming that some relationship among the data in the source and target domains is similar, it focuses on mapping the relational knowledge between the source and target domains. For instance, we can employ this approach to enhance recommendation systems by pooling together rating data from multiple domains and transferring user-item interaction patterns across domains [35]. Essentially, the idea is to build a joint graph that connects users and items in the source and target domains based on their shared attributes, such as language or genre.

2.8 Data Labeling

In this section, we first define and briefly discuss data labeling. Subsequently, we go through a taxonomy of data labeling techniques.

2.8.1 Definition

Data labeling entails the assignment of one or more meaningful and informative labels to data points, which provide the context of learning to a machine learning algorithm. These labels guide the machine learning algorithm to find patterns in the data set that map input attributes to desired outputs by providing supervisory ground truth while training. Moreover, these labels enable the performance evaluation of trained machine learning models against the ground truth.

2.8.2 Taxonomy of Data Labeling Techniques

We can classify data labeling techniques based on the approach into the following categories to better understand the landscape [36].

Manual Labeling

Manual labeling is the simplest and most accurate form of data labeling. A well-known use case is ImageNet which comprises more than 14 million images, hand-annotated with labels indicating the objects in the image [37]. However, ImageNet is an ambitious project that took years to complete, which most machine learning users cannot afford for their specific applications. While common human wisdom may suffice for labeling tasks like annotating generic images with objects in the frame, domain expertise is essential for labeling tasks like annotating X-ray images with malignant tumors. Furthermore, manual labeling is unfeasible in some domains, for instance, manual annotation of cloud system metrics with the corresponding faults is a challenging task, even for domain experts.

Exploiting Existing Labels

It is usually easier and more economical to acquire a small amount of labeled data along with a much larger amount of unlabeled data. Semi-supervised learning techniques enable us to exploit the existing labeled data set to infer the labels for the remaining unlabeled data set. It can be accomplished using inductive or transductive learning. Inductive learning techniques try to discover a general hypothesis from the labeled data points solely that can be applied to any newly obtained unlabeled data points to generate their labels. Transductive learning techniques try to exploit the similarities between labeled and unlabeled data points to deduce the labels for these unlabeled data points. Essentially, transductive

learning techniques do not generate a general hypothesis, and the labeling of any newly obtained unlabeled data points will require the entire algorithm to be executed again.

Crowd-Based Techniques

Though manual labeling is the most accurate method of labeling data instances in most scenarios, it is time and resource expensive and susceptible to human errors and biases. While crowd-based techniques like active learning and crowdsourcing delegate the labeling task to humans, they try to address the shortcomings of manual labeling. Active learning tries to organize the labeling process as a human-machine collaboration to reduce costs, carefully choosing the right data instances to delegate to the supposedly expert crowd for labeling. Crowdsourcing employs many workers for manual labeling who are not necessarily experts while trying to improve the reliability of the labeling process through various mechanisms, acknowledging that the workers may make mistakes.

Weak Supervision

While we always desire to generate accurate labels, it may be too expensive. Alternatively, we may generate large quantities of less accurate data labels, compensating the low quality with the large quantity. In weak supervision, the idea is to generate large quantities of data labels, which are not necessarily as accurate as manually assigned labels. Nevertheless, the large amount of generated labeled data suffices for the inaccuracies within to train a model with reasonable final accuracy. For one, we can generate weak labels programmatically using known patterns or domain heuristics as labeling functions.

2.9 Conclusion

In this chapter, we discussed the background concepts related to this thesis. First, we discussed about cloud computing, beginning with a definition, followed by its essential characteristics, service models, and deployment models, and finishing with a discussion about virtualization. Then, we discussed about faults, defining faults, and describing various types of faults. Subsequently, we discussed about fault detection, defining fault detection, and describing some well-known techniques for fault detection in cloud environments. Then, we discussed about test suites, defining test suites, and describing various types of tests that may comprise a test suite. Then, we discussed about machine learning, beginning with a definition of machine learning, followed by a taxonomy of machine learning techniques, and finishing with a discussion about deep learning. Subsequently, we discussed about active learning, defining active learning, and reviewing a taxonomy of active learning techniques. Subsequently, we discussed about transfer learning, defining transfer learning, and reviewing a taxonomy of transfer learning techniques. Finally, we discussed about data labeling, defining data labeling, and reviewing a taxonomy of data labeling techniques.

Chapter 3

Illustrative Scenario, Requirements, and State of the Art

This chapter first presents a motivating scenario to explain the data labeling problem for fault detection in cloud environments. Subsequently, we use it to derive the requirements of a befitting solution. Finally, we summarize the state of the art data labeling techniques and assess their applicability for labeling cloud metrics data against these requirements.

3.1 Illustrative Scenario

In this section, we present an illustrative scenario to explain the data labeling problem for fault detection in cloud environments. To illustrate a real-time cloud environment, we consider Google's microservices demo application [1], Online Boutique, a web-based e-commerce application, depicted in Fig. 1, which comprises heterogeneous microservices [38] that enable users to browse items, add them to the cart, and purchase them. This application is deployed in a containerized environment by an orchestration platform (e.g., Kubernetes), running on top of virtual machines provisioned from a cloud computing platform (e.g., OpenStack), as depicted in Fig. 2. The microservices would generate a real-time workload on the cloud infrastructure to serve web user requests while meeting the availability requirements and acceptable response time specifications. However, faults at the application, container orchestration platform, or cloud computing platform level may lead to non-compliance with the specifications or even downtimes.

The application, container orchestration platform, and cloud computing platform generate thousands of metrics such as service uptimes, processor usage, memory usage, disk usage, swap usage, average requests, and average response times, which provide insights into the cloud system's availability, health, and performance. For instance, many services in Google's microservices demo application are instrumented with profiling, which measures function call times and counts, memory consumption, processor load, and other resource usage aspects of the application. Kubernetes exposes metrics for its node level components

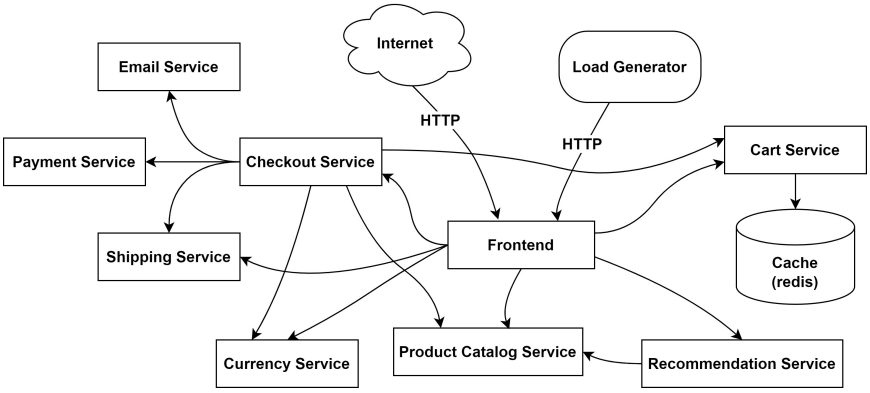


Figure 1: Google's microservices demo application [1].

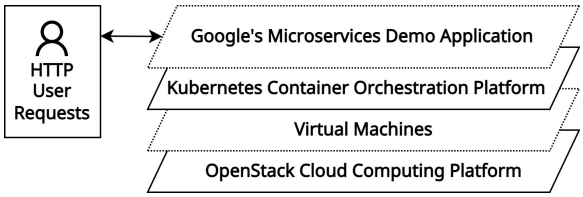


Figure 2: Illustrative scenario of a cloud system.

like Kubelet and Proxy and its control plane components like Application Programming Interface (API) Server, Controller Manager, and Scheduler. OpenStack exposes metrics for its components like Compute, Block Storage, Object Storage, and Networking. While it may be straightforward to acquire a time series of cloud metric data points by employing a cloud monitoring system that periodically scrapes the configured components of the managed cloud system for metrics, it is difficult to label these data points. Manual labeling of these high-dimensional domain-specific time-series metric data points is not only a slow, expensive, and error-prone process, but it is also challenging to manually annotate these metric data points with the corresponding cloud system state, even for domain experts.

Meticulously designed machine learning solutions can identify and learn the correlations between abnormalities in the metrics collected from the cloud system (e.g., unusually high memory usage) and corresponding faults (e.g., memory overload). Although machine learning-based solutions can facilitate the detection of faults, they require a data set of cloud metrics labeled with the corresponding cloud system state for training or evaluation. Most machine learning solutions [11][12][39][40] acquire labeled cloud metrics data using variations of the fault injection mechanism. The fault injection mechanism of labeling entails manually injecting faults into the system and assigning the cloud metrics collected in the fault injection time frame with the injected faults as labels. For instance, in [11], the authors inject recurrent and accumulative faults into the system. Here, recurrent faults involve repetitive induction of CPU stress, HDD stress, or packet loss into the system, followed by subsequent cool-down periods. Accumulative faults include memory stress, HDD stress, or network flooding induced cumulatively into the system in steps over time.

The performance of fault detection solutions relying on the fault injection mechanism of labeled data collection depends on the assumption that the cloud metric data distributions produced by the injected faults satisfactorily model real-world fault scenarios. As deliberate fault injection into a production system can lead to fatal consequences, it is critical to inject faults in a controlled manner. Consequently, fault detection models trained on such data sets might be limited to detecting these constrained artificial faults and fail to capture some data distributions corresponding to real-world fault scenarios. Moreover, automated workload placements according to container resource constraint configurations and overall cluster state in a containerized environment of an orchestration platform, accompanied by real-time workload fluctuations resulting from varying user activity, make the cloud environment highly dynamic. This dynamicity is bound to introduce emerging fault patterns and data or concept drifts in cloud metric data distributions, necessitating retraining over newly acquired data. However, recurrent data acquirement with the fault injection mechanism is impractical, as these long fault injection and system recovery cycles would disrupt cloud production workloads.

3.2 Requirements

From the above discussion with respect to the illustrative scenario, it is clear that we need a befitting mechanism to automatically label cloud metrics with the corresponding cloud system state that meets the following requirements:

- **Requirement 1:** As discussed in Section 3.1, Google's microservices demo application, Kubernetes container orchestration platform, and OpenStack cloud computing platform generate thousands of metrics reflecting the cloud system state. Therefore, the data labeling mechanism should be able to identify and label data distribution patterns in a high-dimensional space of thousands of performance metrics exposed by the cloud system.
- **Requirement 2:** Depending on the given use case, the fault detection solution may need to identify faults at different granular levels. For instance, in the illustrative scenario, we may need to assign a binary system state label of fault or no fault, based on the presence of any fault in any component of the cloud system, in a coarse-grained scenario. On the other hand, we may need to assign finer-grained labels such as normal state, frontend unavailable, checkout service unavailable, and so on, based on Google's microservices demo application's service availabilities. Hence, the data labeling mechanism should be flexible to generate labels of different granularities as required by the use case.
- **Requirement 3:** The data labeling mechanism should be able to assign multiple labels to a metric data instance reckoning with the prospect of multiple concurrent faults in the system. For instance, in the illustrative scenario, if checkout and payment services of Google's microservices demo application are unavailable simultaneously, the

corresponding cloud metric instances should be assigned two labels: checkout service unavailable and payment service unavailable.

- **Requirement 4:** As discussed in Section 3.1, a cloud monitoring system collects a time series of metric data points from the cloud system. Labeling cloud metrics at each time step independently may overlook the temporal information embodied in the time series sequence of cloud metrics data. For instance, in the illustrative scenario, a transient spike in processor consumption in a Kubernetes pod may not manifest into a CPU throttling fault. On the other hand, a consistently high processor consumption in a Kubernetes pod caused by an increased workload on its microservice is likely to manifest into a CPU throttling fault. In order to capture the temporal changes in cloud metrics that manifest into faults, the data labeling mechanism should assign labels to windows of sequential data points from the metric time series rather than individual data points.
- **Requirement 5:** As discussed in Section 3.1, automated workload placements and real-time workload fluctuations in a dynamic cloud environment make it prone to emerging fault patterns and frequent data or concept drifts. Hence, the data labeling mechanism should be able to identify and adapt to emerging metric patterns corresponding to normal and faulty system states.
- **Requirement 6:** The metrics obtained from an application like Google's microservices demo application, a container orchestration platform like Kubernetes, and a cloud computing platform like OpenStack are domain-specific. In addition to being slow, expensive, and error-prone, manual annotation of the high-dimensional domain-specific time-series cloud metric instances with the corresponding cloud system state is challenging, even for domain experts. The fault injection mechanism, an alternative to the manual annotation of cloud metric instances by domain experts, has several inherent limitations, as discussed in Section 3.1. Hence, we need a data labeling mechanism that maps the cloud metric instances with the corresponding system state without requiring any human intervention.

3.3 State of the Art

To further elucidate the rationale for devising a data labeling framework for fault detection in cloud environments, we first review some machine learning-based techniques for fault detection in cloud environments, along with their labeled data acquisition mechanisms. Subsequently, we discuss a taxonomy of data labeling techniques and assess their applicability for labeling cloud metrics data.

3.3.1 Machine Learning-based Fault Detection in Cloud Environments

We can classify the techniques for fault detection in cloud environments into four categories of (i) test suites, (ii) performance metrics monitoring, (iii) system log mining, and

(iv) machine learning. Test suites entail periodic execution of tests, which are typically expensive in terms of computational time and cloud resource consumption. While being comparatively lightweight to test suites, performance metrics monitoring requires a lot of manual configuration to set appropriate metric thresholds for raising alarms. Such configurations also require a lot of domain knowledge to identify the correlation between various metrics and faults. Likewise, system log mining requires an understanding of the correlation between various system log keywords and faults. While being less expensive than test suites, properly designed machine learning solutions can identify and learn the correlations between various performance metrics and faults without requiring a lot of manual configuration efforts and domain expertise like performance metrics monitoring and system log mining.

Supervised machine learning-based solutions aim to derive a function that can map the abnormalities in cloud metrics to corresponding faults by learning from example pairs of cloud metric vectors and supervisory fault labels. In [11], [39], and [40], the authors acquire these example pairs of cloud metric vectors and supervisory fault labels for supervised training using variations of the fault injection mechanism. As discussed in Section 1.2, the fault injection mechanism entails a manual injection of faults into the system followed by assigning the metrics collected in the fault injection time frame with the injected faults as labels. In [11], Soeualhia et al. propose a framework to detect and predict infrastructure-level faults in edge cloud environments using supervised machine learning and statistical techniques. They examine probabilistic models like Markov Chain, Bayesian Network, and Tree-Augmented Naive Bayes, machine learning models like Random Forest, Support Vector Machine, and Neural Network, and statistical models like ARIMA for fault detection. They further employ deep learning models like LSTM and CNN for fault prediction. They inject recurrent and accumulative faults into the system to acquire labeled data. Recurrent faults are characterized by repetitive induction of CPU stress, HDD stress, or packet loss into the system, followed by subsequent cooling down for specific periods. They generate labeled cloud metrics for recurrent faults with system state labels of non-fault, CPU fault, HDD fault, and VM packet loss. Accumulative faults include memory stress, HDD stress, or network flooding induced cumulatively into the system in steps over time. They generate labeled cloud metrics for accumulative faults with system state labels of non-fault and four levels of CPU fault, HDD fault, and VM packet loss (for four cumulative stress induction steps).

In [39], Gupta et al. propose a two-stage supervised deep learning framework for proactive anomaly detection in cloud workloads, implemented as a hybrid of Long Short-Term Memory and Bidirectional Long Short-Term Memory. They try to predict future resource usage and performance metrics patterns in the first stage. The predicted patterns are then classified as normal or abnormal in the second stage. They generate random workload behaviors by stressing CPU, memory, I/O, and disk in different combinations for varying time periods to generate labeled cloud metrics with system state labels: normal and abnormal.

Unsupervised machine learning-based solutions aim to learn probability distributions of cloud metrics corresponding to a normal system state. Cloud metric instances significantly

deviating from these probability distributions can then be identified as corresponding to faults. Although [12] and [41] rely on unlabeled cloud metrics obtained from a fault-free system state for unsupervised training, they still rely on the fault injection mechanism for obtaining the labeled data for evaluation. In [12], Bui et al. propose an unsupervised method for fault detection in multi-tier web applications deployed in cloud computing environments, based on Fuzzy One-Class Support Vector Machine and Exponentially Weighted Moving Average. They further perform fault diagnosis by locating suspicious metrics by applying Random Forest-based Recursive Feature Elimination. For labeled data acquisition, firstly, they generate stress on network traffic, memory, and disk I/O by suddenly increasing the concurrent users accessing the application. Secondly, they trigger an endless loop in the application source code, leading to a stack overflow condition.

In [41], Ghalehtaki et al. propose an unsupervised single/multiple-threshold anomaly detection and explanation method for cloud environments using Long Short-Term Memory Auto Encoder. It relies on normal data obtained from a fault-free system state for training LSTM-AE. It then utilizes the sample and feature reconstruction errors obtained from the trained model to calculate the sample and feature thresholds. Further, they use these thresholds to detect anomalies. They generate CPU faults by stressing CPU resources of the system and network faults by injecting TCP or ping floods into the system to acquire labeled data. Both [12] and [41] generate labeled cloud metrics with system state labels: normal and fault.

On another front, real-time workload fluctuations and automated workload placements in dynamic cloud environments introduce data or concept drifts and emerging fault patterns. Transfer learning is a befitting mechanism for incremental data or concept drift adaptation and emerging fault incorporation in fault detection models while employing previously acquired knowledge for faster learning and improved performance. In [40], Shayesteh et al. propose a fault prediction system for cloud edge environments that automatically adapts to concept drifts via transfer learning mechanism. They train various deep learning models in a supervised fashion and evaluate them for drift adaptation over three transfer learning scenarios: fine-tuning the old model's lower, upper, and all layers. They cumulatively induce CPU stress and network congestion in the system to generate labeled cloud metrics with system state labels of non-fault and four levels of CPU and network faults (for four cumulative stress induction steps).

In a different domain, Li et al. [42] propose a source-supervised deep adversarial transfer learning method for machinery fault detection, which enables the detection of new emerging faults. They train a one-dimensional convolutional neural network employing the adversarial training strategy to classify the data points corresponding to known faults into respective fault classes and detect new emerging fault classes. In [43], Wen et al. propose a supervised deep transfer learning method for fault detection. They perform supervised transfer learning on a three-layered sparse auto-encoder, incorporating the maximum mean discrepancy term in the loss function to minimize the discrepancy penalty between the latent features of source and target data sets.

While supervised machine learning-based solutions [11], [39], and [40] require labeled

cloud metrics data for training, their unsupervised counterparts [12] and [41] require it for evaluation. However, all of them rely on the fault injection mechanism to acquire labeled data, which has several inherent limitations as discussed in Section 3.1. We can adjust the fault injection mechanism based on the use case to obtain a data set of high-dimensional time-series metric data instances labeled with multiple labels according to our granularity requirements. So, it can meet Requirements 1, 2, 3, and 4. Moreover, the fault injection mechanism can be automated to inject faults, collect metrics data from the fault injection time frame, and assign corresponding fault labels to the collected metrics data programmatically, to eliminate human intervention as specified by Requirement 6. However, recurrent data acquirement, necessary for adapting the fault detection models in response to emerging cloud metric data distributions, is practically infeasible using the fault injection mechanism, as the fault injections would disrupt the cloud production workloads. So, it is completely unsuitable to deal with Requirement 5.

3.3.2 Data Labeling Techniques

Although data labeling has not been studied for fault detection in cloud environments, there exists a rich body of literature on data labeling. We can classify data labeling techniques into three main categories of (1) exploiting existing labels, (2) crowd-based techniques, and (3) weak supervision [36].

1. **Exploiting Existing Labels:** The first approach is to exploit the existing (usually small set of) labeled data to infer the labels for the remaining (much larger set of) unlabeled data.

   (a) **Self-Labeled Techniques:** Self-labeled techniques aim to generate labels for the unlabeled data by trusting one's own predictions, following the principle of inductive learning, as discussed in Section 2.8.2. The simplest form of self-labeled techniques is self-training [15]. It involves training a classifier on the labeled data set. This model is then used to make label predictions for the unlabeled data points. The most confident predictions are then added to the labeled data set to retrain the model. This process is performed iteratively until all the unlabeled data points are labeled.

   To exemplify, Yarowsky [15] employs the self-training technique to disambiguate word senses. They exploit the tendency of words to exhibit consistent sense in any given collocation and discourse to tag polysemous words with corresponding senses in a large corpus. Firstly, they create an untagged training data set from the corpus, comprising lines of text with a given polysemous word. Next, they acquire a small number of seed collocations for the polysemous word tagged with the corresponding sense label, for instance, the words life and manufacturing can be seed collocations for two major senses of the word plant. Then, they train a supervised classification algorithm on the seed sets. Subsequently, they apply this classifier to the training data set, take out the training instances tagged with a particular sense

with a probability above a certain threshold, and identify new collocations that co-occur with previous collocations from these newly tagged training instances. They perform this procedure iteratively until the algorithm converges on a stable residual from the training data set. The learned collocations from the final supervised training step may now be applied to annotate new untagged corpus with sense tags.

Another class of self-labeled techniques involves training multiple classifiers on different samples of the labeled data set, for instance, tri-training [44]. Tri-training initially trains three models on the labeled data set using bagging-based ensemble learning. In each round of tri-training, some unlabeled data points are labeled and added to the training data set of a model if the other two models agree on the label prediction for that data point. Each model is updated iteratively until no model changes further. Finally, the unlabeled data points are labeled using majority voting. Since tri-training uses majority voting, no explicit confidence measures for the model predictions are required, which imparts it better generalization ability.

Another class of self-labeled techniques uses multiple learning algorithms to train different classifiers on the same labeled data set, for instance, democratic co-learning [45]. Labels are generated for the unlabeled data using weighted voting over the predictions of these models. Newly labeled data points are then added to the training data set of the models whose label predictions differ from the majority. Each model is updated iteratively until no more data is added to the training data set of a model. Since different machine learning algorithms have different biases, their results can be combined to produce better predictions.

Another class of self-labeled techniques uses multiple views, which are feature subsets that are sufficient for learning but conditionally independent given the class, for instance, co-training [46]. A model is trained for each feature set and is used to teach the other model trained on another feature set. The goal is to minimize the classification error by maximizing the prediction agreement of the models over the unlabeled data points. However, it may not always be feasible to partition the feature set into such subsets.

Self-labeled techniques can be adapted to label high-dimensional time-series metric data instances with multiple labels based on the use case, provided we have or can acquire a labeled sample of such data instances to train an initial classifier or classifiers. Moreover, self-labeled techniques can generate labels at the desired granularity, provided we have labels at the same or more granular level in the sample data set. So, they can meet Requirements 1, 2, 3, and 4. Furthermore, they do not require human intervention as specified by Requirement 6. Self-labeled techniques can generate data labels for the unlabeled data instances following similar data distribution as the labeled data with satisfactory accuracy. However, they are unsuitable for labeling cloud metrics data, which is subject to emerging data distributions. Hence, they do not satisfy Requirement 5.

(b) **Graph-Based Label Propagation Techniques:** Graph-based label propagation techniques like [47], [16], and [48] try to derive a weighted graph structure among the data points based on their similarities. Then they exploit this graph structure to

propagate labels from the labeled data points to the unlabeled data points. Graph-based label propagation techniques follow the principle of transductive learning, as discussed in Section 2.8.2.

To exemplify, Blum et al. [47] propose a graph-based label propagation technique based on finding minimum cuts in graphs, discussing it in the setting of binary classification. They arrange the labeled and unlabeled data points as vertices in a weighted graph structure where the edges connecting the data points are weighted based on the similarity or distance between them, specifically experimenting with the k-nearest neighbors algorithm and ϵ-neighborhood. Furthermore, they add a single source node $v+$ connected with infinite weight to all positively labeled data points, and a single sink node $v-$ connected with infinite weight to all negatively labeled data points. Subsequently, determining the minimum cut corresponds to finding a set of edges with the minimum combined weight that, when removed, results in a graph with no paths from the source node to the sink node. Finally, all unlabeled vertices directly or indirectly connected to $v+$ are labeled positively and all unlabeled vertices directly or indirectly connected to $v-$ are labeled negatively.

Graph-based label propagation techniques follow the principle of transductive learning, which requires all unlabeled data instances to be labeled to be available upfront. Thus, they typically require executing the entire algorithm again to label any newly encountered data instances. Therefore, applying these techniques to streaming time-series metrics data is challenging, and their applicability is limited with regard to Requirement 4. We can adjust graph-based label propagation techniques to assign high-dimensional metric data instances with multiple labels of desired granularity based on the use case, provided we have labels at the same or more granular level in the available labeled data set. So, they can meet Requirements 1, 2, and 3. Furthermore, they do not require human intervention as specified by Requirement 6. Graph-based label propagation techniques can generate data labels for the unlabeled data instances following similar data distribution as the labeled data with satisfactory accuracy by exploiting the similarity or distance between them. However, they are unsuitable for labeling cloud metrics data, which is subject to emerging data distributions. Hence, they do not satisfy Requirement 5.

2. **Crowd-Based Techniques:** Though manual labeling is the most accurate method of labeling data instances in most scenarios, it is time and resource expensive and susceptible to human errors and biases. While crowd-based techniques delegate the labeling task to humans, they try to address the shortcomings of manual labeling.

   (a) **Active Learning:** Active learning tries to organize the labeling process as a human-machine collaboration to reduce costs, carefully choosing the right data instances to delegate to the crowd for labeling. It usually involves one or more machine learning models to make label predictions for the unlabeled data points. Based on these label predictions, we identify the data points whose labels will be most informative to the underlying machine learning algorithms and delegate them to the expert crowd for labeling. This crowd-labeled data helps the underlying algorithms improve

progressively. Uncertainty sampling [13], which chooses the next unlabeled data point with the highest model prediction uncertainty is the simplest approach to active learning. Another approach is margin sampling, which chooses the next unlabeled data point where the prediction probability difference between the first and second most probable label is the smallest. Yet another approach is query-by-committee [49], which extends uncertainty sampling by training a committee of models on the same labeled data set. Here, the data point where most of the models disagree on their predictions is considered to be the most informative.

To exemplify, Lewis et al. [13] propose uncertainty sampling, an algorithm for sequential sampling of data instances for manual labeling during machine learning of statistical classifiers, specifically focusing on the problem of text categorization. They employ a probabilistic classifier that estimates the probability of a data instance belonging to a particular text category given an observed pattern within the text, initially trained on a small labeled data set. After each training iteration, the classifier makes probabilistic class predictions for each unlabeled data instance. As a probability of 0.5 corresponds to the classifier being most uncertain of the class label, they chose $b/2$ data instances with predicted probabilities closest to 0.5 and above it and $b/2$ data instances with predicted probabilities closest to 0.5 and below it to create a set of total b data instances to be delegated for manual labeling. This method reduces the selection of exact duplicates for delegation. On acquiring new manually labeled data instances, they train a new classifier over the entire labeled data set. They perform this process iteratively to improve the classifier gradually while reducing the manual labeling effort.

As discussed, active learning techniques require domain expert labelers to label the informative data instances. However, manual annotation of high-dimensional time-series cloud metrics data with the corresponding faults is a challenging task even for domain experts. Hence, the applicability of conventional active learning techniques is limited in terms of Requirements 1 and 4. In general, active learning techniques can support multi-labeling tasks and flexible label granularities in consonance with Requirements 2 and 3. Moreover, the capability of active learning systems to progressively improve enables them the keep up with emerging data distributions in consonance with Requirement 5. Since active learning techniques involve a human component, they do not satisfy Requirement 6.

(b) **Crowdsourcing:** Another crowd-based technique is crowdsourcing, which in contrast to active learning, leans toward employing many workers for manual labeling who are not necessarily experts. So, crowdsourcing techniques are most befitting to labeling tasks where general human wisdom produces accurate results, for instance, image annotation and text categorization. The literature on crowdsourcing [14], [50], [51], [52], [53], and [54] tries to improve the reliability of the labeling process, acknowledging that the workers may make mistakes. For instance, crowdsourcing generally involves laying out comprehensive guidelines to crowd workers to prevent differing interpretations of concepts and inconsistent labels. However, deriving such guidelines to cover all the nuances and subtleties in a data set would

require a close examination of much of the data, which is typically infeasible in crowdsourcing settings. Revolt [52], which needs only the top-level concept of interest, eliminates the burden of creating detailed label guidelines by harnessing crowd disagreements for identifying ambiguous concepts and creating groups of semantically related items used for post hoc judgments to define the final label decision boundaries.

At a high level, Revolt divides a dataset into multiple batches and then coordinates crowd workers in small teams of three through a labeling process comprising three stages: Vote, Explain, and Categorize. The Vote stage collects independent label judgments for the same item from multiple crowd workers in the same group. Once all crowd workers have made their judgments, items receiving unanimous labels from crowd workers in the group are assigned final labels. On the contrary, items receiving conflicting labels from crowd workers in the group proceed to the Explain Stage. In the Explain stage, crowd workers provide short explanations, describing their rationale for each label to the rest of the group for items flagged as uncertain in the previous stage. Next, in the Categorize stage, crowd workers group uncertain items into categories while considering explanations from others in the group. They can either select from an existing list of categories or add a new one. After collecting crowd feedback for all batches, Revolt assigns labels to certain items. Furthermore, it creates structures of uncertain items by applying simple majority voting on the category names suggested by crowd workers for each item. Finally, label requesters review these structures and make final label decisions for them.

Manual annotation of high-dimensional time-series cloud metrics data with the corresponding faults is a challenging task even for domain experts, hence unfeasible for crowd workers. Hence, crowdsourcing does not meet Requirements 1 and 4. Crowdsourcing can support multi-labeling tasks and flexible label granularities in consonance with Requirements 2 and 3. Since crowdsourcing is a slow and expensive process, it is unsuitable for recurrent data acquirement to keep up with emerging data distributions. Thus, it is not a suitable alternative to address Requirement 5. Lastly, since crowdsourcing involves a human component, it does not satisfy Requirement 6.

3. **Weak Supervision:** The third approach to data labeling is weak supervision, where the idea is to generate large quantities of data labels, which are not necessarily as accurate as manually assigned labels. Nevertheless, the large amount of generated labeled data suffices for the inaccuracies within to train a model with reasonable final accuracy. One form of weak supervision is data programming, as in the Snorkel system [17]. In Snorkel, instead of manually labeling training data, users specify multiple programmable labeling functions expressing weak supervision sources like known patterns, domain heuristics, and external knowledge bases, which can have unknown accuracies and correlations. Next, these labeling functions are combined into a generative model, learning from the agreements and disagreements of the labeling functions without any

ground-truth data. This step also allows Snorkel to estimate the accuracies and correlations among the labeling functions. The generative model is essentially a re-weighted combination of the user-provided labeling functions, which tends to be precise but with low coverage. However, it outputs a set of probabilistic labels that can be used to train a discriminative model. Finally, a noise-aware discriminative model is trained using the labels produced by the generative model to generalize beyond the labeling functions and increase the coverage and robustness on unseen data. However, data programming requires ample domain knowledge to define labeling functions. Moreover, weakly supervised learning typically has comparatively inferior generalization and accuracy.

We can adapt weak supervision techniques to label high-dimensional time-series metric data instances with multiple labels of desired granularities based on the use case to satisfy Requirements 1, 2, 3, and 4. Furthermore, they do not require human intervention as specified by Requirement 6. However, weak supervision techniques do not inherently extend to accommodate emerging data distributions. Thus, they do not satisfy requirement 5.

Table 1: Summary of Evaluation of Data Labeling Techniques.

	Requirements					
Data Labeling Techniques	**1**	**2**	**3**	**4**	**5**	**6**
Fault Injection Mechanism	✓	✓	✓	✓	✗	✓
Self-Labeled Techniques	✓	✓	✓	✓	✗	✓
Graph-Based Label Propagation Techniques	✓	✓	✓	✻	✗	✓
Conventional Active Learning	✻	✓	✓	✻	✓	✗
Crowdsourcing	✗	✓	✓	✗	✗	✗
Weak Supervision	✓	✓	✓	✓	✗	✓

We denote the requirements met out of the box or with apparent adaptations by ✓, requirements not met by ✗, and requirements met with apparent limitations by ✻

Table 1 shows a summary of the evaluation of the data labeling techniques discussed in this section, along with the fault injection mechanism discussed in Section 3.3.1, against the requirements presented in Section 3.2. Although there is extensive research on fault detection in cloud environments, the data labeling aspects of the problem have not received due attention. Moreover, none of the data labeling techniques in the literature is inherently suitable for labeling cloud metrics data for fault detection.

Chapter 4

Proposed Framework

This chapter presents the proposed test suite-based active learning framework for labeling cloud metrics data with the corresponding cloud system state while meeting the requirements defined in Section 3.1. We begin with a detailed description of various components of the framework. Subsequently, we describe the workflow of the framework in action, putting these components together. Finally, we conclude with a review of the proposed framework in light of the requirements defined in Section 3.1.

4.1 Framework Components

Fig. 3 presents an overview of the proposed framework. It comprises (A) a managed cloud system, (B) a cloud monitoring system, (C) a data preprocessing module, (D) an active learning classifier, (E) active learning informativeness estimators, (F) an active learning oracle test suite, and (G) an informative data instances buffer. In this chapter, we elaborate on each of these components in detail.

4.1.1 Managed Cloud System

The managed cloud system is a cloud platform with a deployed workload consuming the cloud resources to provide the desired services while meeting their specifications. For instance, Google's cloud-native microservices demo application, Online Boutique, deployed in a Kubernetes environment running on top of virtual machines provisioned from an OpenStack cloud. The microservices would generate a real-time workload on the cloud to serve web user requests while maintaining availability and meeting acceptable response time specifications.

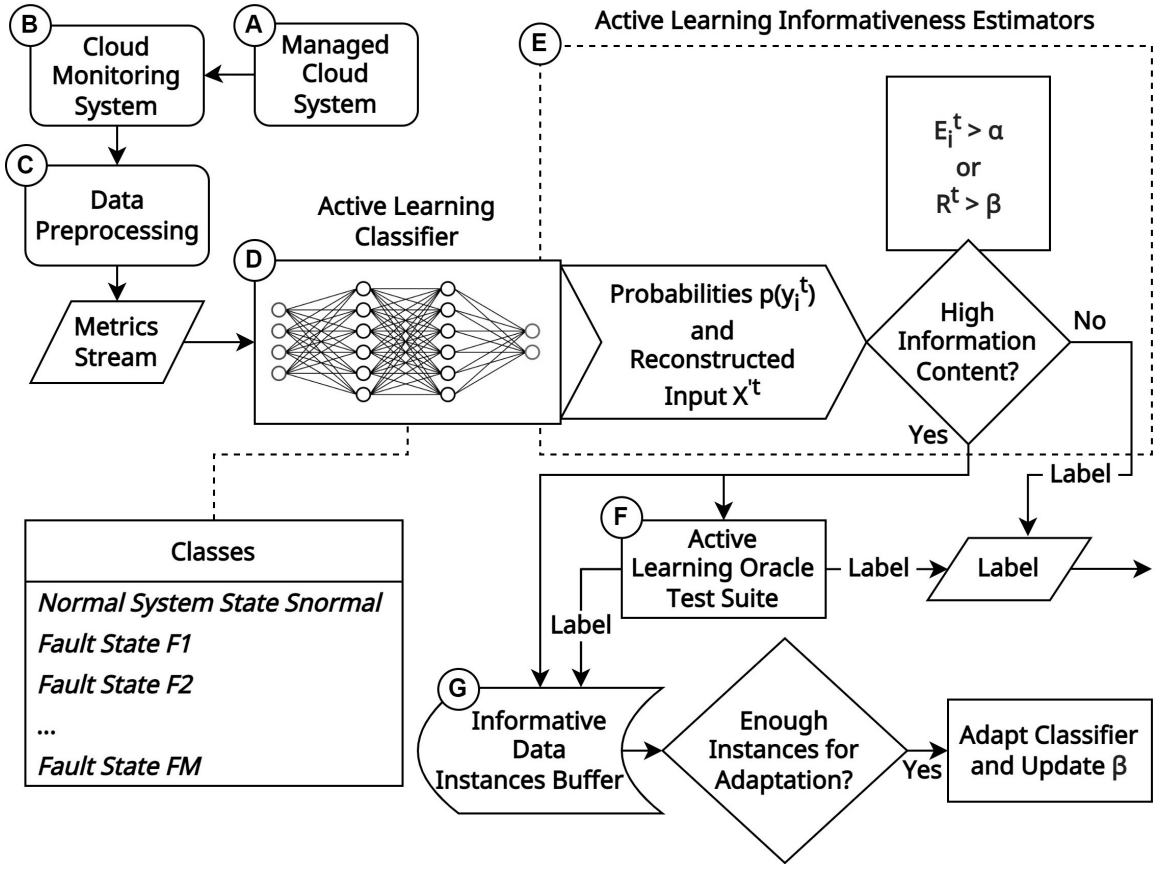


Figure 3: Overview of the proposed framework.

4.1.2 Cloud Monitoring System

The managed cloud system is instrumented with a cloud monitoring system that collects metrics from configured components of the managed cloud system at specified intervals. For instance, it can scrape Kubernetes in the managed cloud system for metrics from its node components like kubelet and proxy and control plane components like API server, controller manager, and scheduler.

4.1.3 Data Preprocessing

The data preprocessing module transforms raw metric data points into a more suitable form for a performant application of a machine learning algorithm. In the remainder of this section, we discuss some of the quintessential preprocessing operations for metrics obtained from a cloud monitoring system.

We usually obtain two types of metrics from a cloud monitoring system, which are gauge and counter. A gauge-type metric has a numerical value that can arbitrarily go up and down over time. On the other hand, a counter-type metric has a cumulative value that

increases monotonically over time or is reset to zero on restart. As a counter-type metric is not suitable for learning in its raw form, firstly, we transform it into a gauge-type metric. We achieve this by replacing the value of a counter-type metric at any timestamp with its difference from its value at the previously recorded timestamp.

The metrics obtained from a cloud monitoring system usually have values on different scales. Therefore, we need to bring all the metrics on the same scale so that the machine learning algorithm is not unfairly biased towards metrics with generally higher values. This is done by normalizing the metrics to the range [0, 1] using MinMaxScaler. The MinMaxScaler transformation for a metric value m of a metric f can be defined as:

$$m_{scaled} = \frac{(m - f_{min})}{f_{max} - f_{min}} \tag{1}$$

where f_{min} and f_{max} are the minimum and maximum values of the metric f respectively.

Finally, we apply a sliding window over the metric time series to obtain input metric data instances for time series classification that consolidate system state information over a period rather than an instant of time. The resulting metric data instance is denoted by X^t, where t is the timestamp of the last metric data point in the sliding window. Further, we denote each element of X^t by x^t_{ij}, where $i \in \{1, 2, ..., L\}$, $j \in \{1, 2, ..., F\}$, L is the size of the pre-processing sliding window, and F is the number of metrics in each metric data point.

4.1.4 Active Learning Classifier

The managed cloud system can produce a multitude of metrics with diverse characteristics and embodied information. The application of traditional machine learning algorithms over these metrics entails domain expertise and efforts to engineer the optimum features for learning. Moreover, the features engineered are task-specific and may not transfer well to other tasks. On the contrary, deep learning algorithms can self-learn hierarchical feature representations, alleviating the need for domain-expert feature engineering. Additionally, the learned feature representations are better transferable, facilitating transfer learning of emerging fault patterns and data or concept drifts. Hence, a deep learning classifier is a good fit for our framework.

The cloud monitoring system produces a time series of metrics that reveals the temporal changes in the managed cloud system that could manifest into faults. An elementary approach is to classify the metrics at each point in time separately, but it overlooks the information embodied in the temporal sequence of the metrics. As it is of essence that the classifier learns this temporal dynamic behavior to identify the system states precisely, we need to employ a time series classifier.

At any point in time, the managed cloud system can either be in a normal system state or a system state with one or more faults. The prospect of multiple faults occurring simultaneously necessitates the use of a multi-label classifier to annotate X^t with multiple fault

labels. Detection of multiple concurrent faults is a challenging problem owing to the unavailability of data for such scenarios, which makes it indispensable to capture and label such data instances.

In congruence with the abovementioned requisites, we employ a multi-label time-series deep learning classifier as our active learning classifier to return probabilities of X^t associated with the following classes:

- Normal System State S_{normal}, which corresponds to a normally functioning state of the cloud system.
- Fault State F_k, where $k \in \{1, 2, ..., M\}$ correspond to one of the M fault categories. For instance, we can consider F_k to be a CPU over-utilization fault in one of the pods in the illustrative managed cloud system, such as Frontend CPU Fault.

Let $Y'^t = \{p(y^t_{S_{normal}}), p(y^t_{F_1}), p(y^t_{F_2}), ..., p(y^t_{F_M})\}$ be the output of the classifier, where $p(y^t_i)$ is the predicted probability of X^t belonging to class i, $\forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$.

We employ a deep neural network model to realize the active learning classifier, as shown in Fig. 4, with the following architecture.

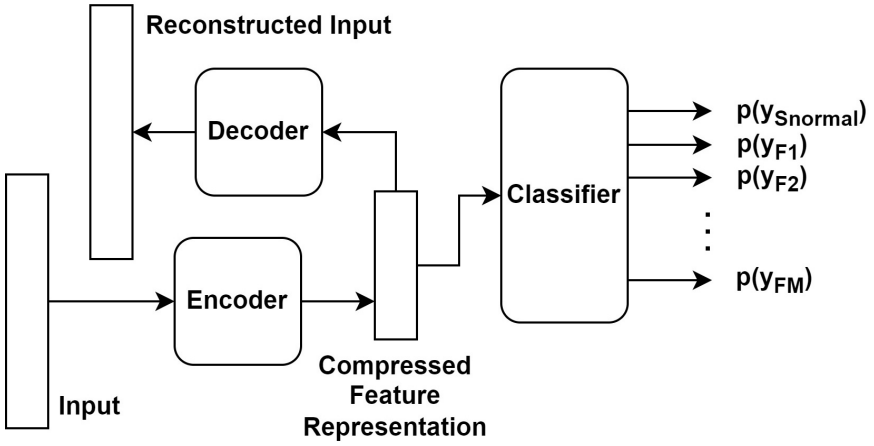


Figure 4: Overview of the proposed active learning classifier.

- Encoder $E(\mathrm{X}^t)$ is a Long Short-Term Memory (LSTM) neural network for generating a compressed feature representation z^t of dimension D for a metric data instance X^t lying in a high dimensional metric space.
- Decoder $D(z^t)$ is an LSTM neural network to map the compressed feature representation z^t back to the input metric data instance X^t.
- Classifier $C(z^t)$ is a feed-forward neural network to generate probabilities of the compressed feature representation z^t belonging to each of the classes.

The training procedure is split into two parts of training the Encoder-Decoder network and training the Encoder-Classifier network.

The training of the Encoder-Decoder network aims to minimize the reconstruction error $Loss_D$ defined by root mean square error, which is the normalized distance between input X^t and its reconstructed value $\mathrm{X}^{'t}$. Essentially, it is the square root of the average squared reconstruction error over elements $x_{ij}^{'t}$ of $\mathrm{X}^{'t}$. While squaring the errors helps penalize larger errors more than smaller ones, it also squares the unit of error. Taking the square root of average squared errors rescales the error back to its original unit. It is defined as follows:

$$Loss_D = \left\{ \frac{1}{L \cdot F} \sum_{i=1}^{L} \sum_{j=1}^{F} (x_{ij}^{t} - x_{ij}^{'t})^2 \right\}^{\frac{1}{2}} \tag{2}$$

where x_{ij}^{t} is a metric's original and $x_{ij}^{'t}$ is its reconstructed value at timestamp t.

The Classifier generates the probability measures of z^t belonging to classes S_{normal} and F_k, each comprising an independent binary classification problem and altogether comprising a multi-label classification problem. Therefore, we employ binary cross-entropy loss, which is the most commonly used loss function for binary classification problems. It provides a measure of the difference between the predicted and the actual probability distributions. The Encoder-Classifier network training aims to minimize the binary cross-entropy loss $Loss_C$ defined as follows:

$$Loss_C = -\frac{1}{K+1} \sum_{i=1}^{K+1} \{ y_i^t \cdot \log_2(p(y_i^t)) + (1 - y_i^t) \cdot \log_2(1 - p(y_i^t)) \} \tag{3}$$

where $K+1$ is the total number of output classes and $y_i^t \in \{0, 1\}$ is the label indicating if X^t belongs to class i. For each X^t, the label set $Y^t = \{y_{S_{normal}}^t, y_{F_1}^t, y_{F_2}^t, ..., y_{F_M}^t\}$.

We train the Encoder-Decoder and Encoder-Classifier networks simultaneously to minimize $Loss_C + Loss_D$ for a given batch of input samples. While the Encoder-Classifier network performs the crucial task of giving out class probabilities, the Encoder-Decoder network ensures that the learned feature representation incorporates all characteristics of the input samples to ensure better transferability to emerging data distributions down the line. Furthermore, the Encoder-Decoder network helps identify input samples from emerging data distributions, as discussed in Section 4.1.5.

4.1.5 Active Learning Informativeness Estimators

We follow the active learning approach of selecting only the most informative samples from the metric data space to be labeled by the test suite. Though the test suite can identify the faults leading to service specification failures with high accuracy, they tend to be slow and cloud resource-expensive. So, we establish a mechanism where the test suite is only triggered if an informative metric data instance is identified based on the following

informativeness estimators.

- The uncertainty associated with the predicted probability of class i, $\forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$, is above a pre-configured threshold α. We quantify the uncertainty over class i at timestamp t as the mean binary entropy E_i^t over a window of H data instances given by:

$$E_i^t = -\frac{1}{H} \sum_{n=(t-H+1)}^{t} \{p(y_i^n) \cdot \log_2(p(y_i^n)) + (1 - p(y_i^n)) \cdot \log_2(1 - p(y_i^n))\} \quad (4)$$

  where $p(y_i^n)$ is the predicted probability of the data instance at timestamp n belonging to class i.

  As the Classifier produces probabilistic scores, the final classifications depend on the probability threshold configurations. Let T_u and T_l denote the high positive and low negative classification thresholds, where $T_l = 1 - T_u$. We infer that a data instance with a probability score greater than T_u on a class belongs to that class and smaller than T_l on a class does not belong to that class. On the other hand, a probability score between T_u and T_l indicates a high classification uncertainty. Consequently, we set α to trigger the test suite for labeling data instances with high classification uncertainties, defined as:

$$\alpha = -\{T_u \cdot \log_2(T_u) + T_l \cdot \log_2(T_l)\} \quad (5)$$

  We note, however, that transient occurrences of probability scores between T_u and T_l for a class may unnecessarily trigger the test suite. To ensure that the test suite triggers only when the classification uncertainties are constantly high for a class, we take the mean of binary entropies over a window of H data instances for that class as our informativeness estimator. In case of transient occurrences of probability scores between T_u and T_l for a class, we infer the corresponding class label for data instance X^t based on the class label assigned to data instance X^{t-1} at the previous timestamp.

- The reconstruction error is greater than the pre-configured threshold β. The reconstruction error R^t of data instance X^t recorded at timestamp t is given by:

$$R^t = \left\{ \frac{1}{L \cdot F} \sum_{i=1}^{L} \sum_{j=1}^{F} (x_{ij}^t - x_{ij}^{'t})^2 \right\}^{\frac{1}{2}} \quad (6)$$

  Threshold β can be configured as the highest reconstruction error produced by the trained Encoder-Decoder network over the data instances used for training. The Encoder-Decoder network produces higher reconstruction errors for data instances from unseen data distributions, thus helping discover unseen data distributions.

Binary variable Z^t determines whether or not the test suite is triggered for the input sample

X^t:

$$Z^t = \begin{cases} 1, & \text{if } E_i^t > \alpha \\ 1, & \text{if } R^t > \beta \\ 0, & \text{otherwise.} \end{cases}$$

4.1.6 Active Learning Oracle: Test Suite

A test suite is a set of tests, primarily functional tests, intended to programmatically identify the faults in the managed cloud system in real-time. For instance, in our use case explained in Section 3.1, the functional tests would identify the services that fail to comply with the availability and acceptable response time specifications. To further narrow down to the root cause of the service failures, we can define fault identification tests intended to identify the factors that caused the non-conformance to the specifications. These functional and fault identification tests can qualify the metric data instances with labels such as Frontend CPU Fault, and Recommendation Service CPU Fault. In another scenario, standard test suites like Kubernetes E2E [55], which tests the end-to-end behavior of Kubernetes components, ensuring consistent and reliable behavior of core Kubernetes functionalities, can be integrated with the proposed framework to identify Kubernetes cluster level faults.

Once an informative data instance is identified using our informativeness estimators defined above, we store the current and subsequent data instances in the informative data instances buffer and trigger the test suite. By running the test suite, we identify the real-time system state and accordingly generate a label set for the data instances in the buffer acquired during the run-time of the test suite. We execute the test suite iteratively until we obtain enough labeled data instances to adapt the active learning classifier. Subsequently, we augment the newly acquired labeled data instances with the existing training data set. We use the transfer learning mechanism to adapt the active learning classifier over the augmented training data set. Specifically, it involves fine-tuning the existing Encoder and Decoder networks and training a newly spawned Classifier network while discarding the existing Classifier network. Using transfer learning over previously learned latent feature representations to incorporate emerging data distributions enables efficient and faster adaptation. Finally, we update the threshold β with the highest reconstruction error produced by the adapted Encoder-Decoder network over the augmented training data set.

4.2 Workflow of the Framework

In this chapter, we discuss the workflow of the proposed framework, describing how its components interact and work together to produce the data labels, as depicted in Fig. 5. We begin by describing the initial setup of the framework, followed by the subsequent operation of the framework to label live metrics data.

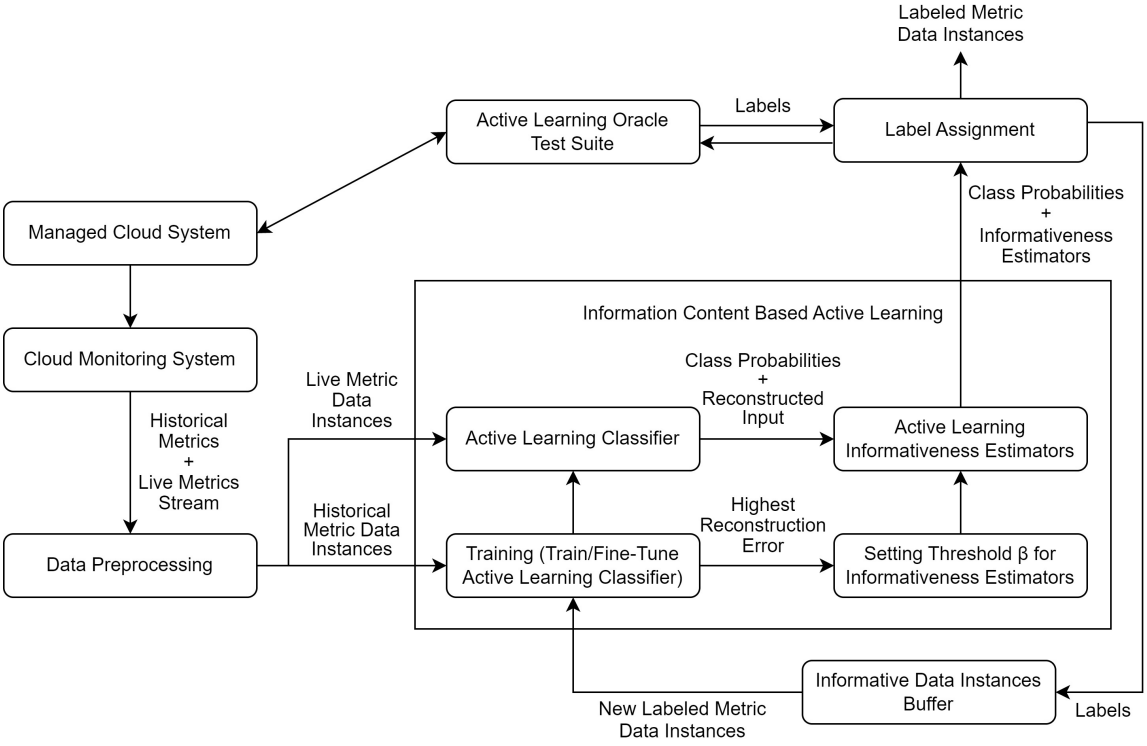


Figure 5: Workflow of the proposed framework.

4.2.1 Initial Setup

As a starting point to deploy the framework, we need to initialize and configure various components of the framework as described in the following.

- Firstly, we need to acquire a sample of labeled historical metrics data, most importantly, for the initial training of the active learning classifier. This labeled training data set should be representative of each class i, $\forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$. The simplest approach to acquire the labeled training data for class F_i, $\forall F_i \in \{F_1, F_2, ...F_M\}$ is to use the fault injection mechanism. Essentially, we manually inject fault F_i into the managed cloud system and manually annotate the metrics data collected by the cloud monitoring system in the fault injection time frame with the injected fault F_i as the label. On the other hand, we can easily acquire the labeled training data for class S_{normal} by manually annotating the metrics data collected by the cloud monitoring system in a time frame where the managed cloud system is operating normally without any faults with S_{normal} as the label. It is important to note that this is a one-time activity required to initialize the framework, which can be skipped if we already have a labeled training data set available.

- Next, we feed this labeled training data set to the data preprocessing module to apply the transformations discussed in Section 4.1.3. First, we convert each counter-type metric

in the training data to a gauge-type metric. Second, we need to normalize the training data. Here, we define the MinMaxScaler over the training data. Then, we normalize the training data using this MinMaxScaler. Third, we apply the pre-processing sliding window over the training data to obtain the final training data instances for time series classification.

- Next, we employ this transformed labeled training data set to train the initial active learning classifier, that is, the initial Encoder, Decoder, and Classifier networks, as discussed in Section 4.1.4.

- Finally, we set appropriate thresholds, α and β, for the active learning informativeness estimators as discussed in Section 4.1.5. While α is derived from the manually configured classification probability thresholds, T_u and T_l, as described in Equation 5, β is derived from the training data set and trained Encoder-Decoder network. Essentially, we employ the trained Encoder-Decoder network to reconstruct each training data instance, record corresponding reconstruction errors as described in Equation 6, and set β equal to the highest reconstruction error produced.

4.2.2 Workflow

In the following, we describe the workflow of the framework to label live metrics data after the initial setup.

- The cloud monitoring system periodically scrapes the managed cloud system to collect metric data points from the configured components of the managed cloud system. We feed this live stream of metric data points to the data preprocessing module.

- When the data preprocessing module encounters a new metric data point, it transforms it by converting the counter-type metrics to gauge-type and normalizing the values of individual metrics in the metric data point, as discussed in Section 4.1.3. Subsequently, the pre-processing sliding window slides by one place to create a new metric data instance X^t incorporating the latest acquired metric data point. Subsequently, we feed this latest metric data instance X^t to the active learning classifier module.

- As discussed in Section 4.1.4, in the active learning classifier module, the Encoder-Classifier network produces the class probabilities $Y^{'t}$=$\{p(y^t_{S_{normal}}), p(y^t_{F_1}), p(y^t_{F_2}), ..., p(y^t_{F_M})\}$ for X^t. Furthermore, the Encoder-Decoder network produces the reconstructed value $\mathrm{X}^{'t}$ for X^t. Subsequently, we feed this metric data instance X^t, the corresponding class probabilities $Y^{'t}$, and the corresponding reconstructed value $\mathrm{X}^{'t}$ to the active learning informativeness estimators module.

- As discussed in Section 4.1.5, we calculate the mean binary entropy E^t_i and the reconstruction error R^t for X^t. If $E^t_i > \alpha$ or $R^t > \beta$, we consider X^t as informative and set $Z^t = 1$, otherwise we set $Z^t = 0$.

- If $Z^t = 0$, the label assignment module assigns labels to X^t based on the class probabilities Y'^t and the classification probability thresholds T_u and T_l as follows.
  - If $p(y_i^t) >= T_u, \forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$, it assigns X^t with the label i.
  - If $p(y_i^t) <= T_l, \forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$, it does not assign X^t with the label i.
  - If $T_l < p(y_i^t) < Tu, \forall i \in \{S_{normal}, F_1, F_2, ...F_M\}$, it assigns X^t with the label i if X^{t-1} was assigned with the label i, otherwise not.
- If $Z^t = 1$, the label assignment module triggers the test suite to label X^t and subsequent metric data instances until we acquire enough labeled data instances for adapting the active learning classifier as follows.
  - The test suite executes on the managed cloud system to identify its state in real time and generates a label set accordingly. We store X^t and subsequent metric data instances acquired during the run time of the test suite in the informative data instances buffer and assign them with the label set returned by the test suite.
  - We continue to execute the test suite iteratively to label the incoming metric data instances until we obtain enough labeled metric data instances to adapt the active learning classifier. Subsequently, we augment the newly acquired labeled data instances with the existing labeled training data set and begin the adaptation process.
  - We use the transfer learning mechanism to adapt the active learning classifier over the augmented training data set. Specifically, we fine-tune the existing Encoder and Decoder networks to transfer the previously learned parameters for an efficient and faster adaptation, and train a newly spawned Classifier network while discarding the existing Classifier network, using the training procedure discussed in Section 4.1.4.
  - Finally, we update the threshold β for the active learning informativeness estimators module with the highest reconstruction error produced by the adapted Encoder-Decoder network over the augmented training data set.

4.3 Conclusion

In this chapter, we presented a detailed description of the proposed test suite-based active learning framework for labeling cloud metrics data, beginning with a discussion of the crucial components of the framework, followed by an explanation of the overall operation of the proposed framework with these components put together in action. The proposed framework adopts the concept of active learning to identify and adapt to emerging metric data distributions, which may be emerging fault patterns or data or concept drifts, to address Requirement 5. It replaces the expert human labeler in conventional active learning settings with a test suite as the active learning oracle to eliminate human intervention, in consonance with Requirement 6. We employ a multi-label time-series deep learning classifier as our active learning classifier to return probabilities of high-dimensional time-series

metric data instances belonging to a normal system state or a system state with one or more faults, addressing Requirements 1, 2, and 4. If the active learning classifier makes confident predictions for a metric data instance, we use these predictions to assign data labels to this metric data instance. On the contrary, if the predictions are highly uncertain, we trigger a test suite to identify the system state in real time and generate the labels. Subsequently, we exploit these metric data instances labeled with the help of the test suite to adapt the active learning classifier so that it can confidently label metric data instances from a similar data distribution in the future. By leveraging this active learning mechanism, we attain a trade-off between cloud resource consumption by the test suite and labeling efficacy. Moreover, we adopt the concept of transfer learning to facilitate efficient and faster adaptation of the active learning classifier. Lastly, we can adjust the framework to generate data labels at various granularities, as specified in Requirement 3, by defining appropriate test suites to assign data labels according to our granularity requirements.

Chapter 5

Performance Evaluation

In this chapter, we evaluate the performance of our proposed data labeling framework on a cloud testbed resembling a real-world scenario. We begin with a description of our experimental setup along with the configurational settings used for various framework components. Subsequently, we present and analyze the results of our experiments for various emerging data distribution scenarios.

5.1 Experimental Setup

In the following, we describe our lab setup, configurational parameter settings, employed test suite, programming tools used, and fault injections.

5.1.1 Lab Setup

Figure 6 illustrates our experimental setup, which comprises 4 virtual machines, *master*, *worker-1*, *worker-2*, and *admin*, each with Intel Xeon E312xx 4 CPU cores and 4 GB RAM. These virtual machines are provisioned from Ericsson Research's OpenStack private cloud Xerces. We use a separate physical machine with Intel(R) Core(TM) i5-9300H 2.40GHz CPU and 8 GB RAM to train and adapt the active learning classifier. We deploy a Kubernetes cluster over these virtual machines, where *master* acts as Kubernetes master node, *worker-1*, *worker-2*, and *admin* act as Kubernetes worker nodes. We deploy Google's microservices demo application on this cluster configured to run its pods on *worker-1* and *worker-2*. The *load generator* generates a real-time workload on the cluster by continuously sending HTTP requests to the application's *frontend*, imitating realistic user shopping flows. All these components make up our managed cloud system.

We employ Prometheus to scrape the cluster for cAdvisor [2] and kube state metrics [3] every 5 seconds. While cAdvisor exposes resource usage and performance data from running containers, kube state metrics provide information about the state of Kubernetes objects

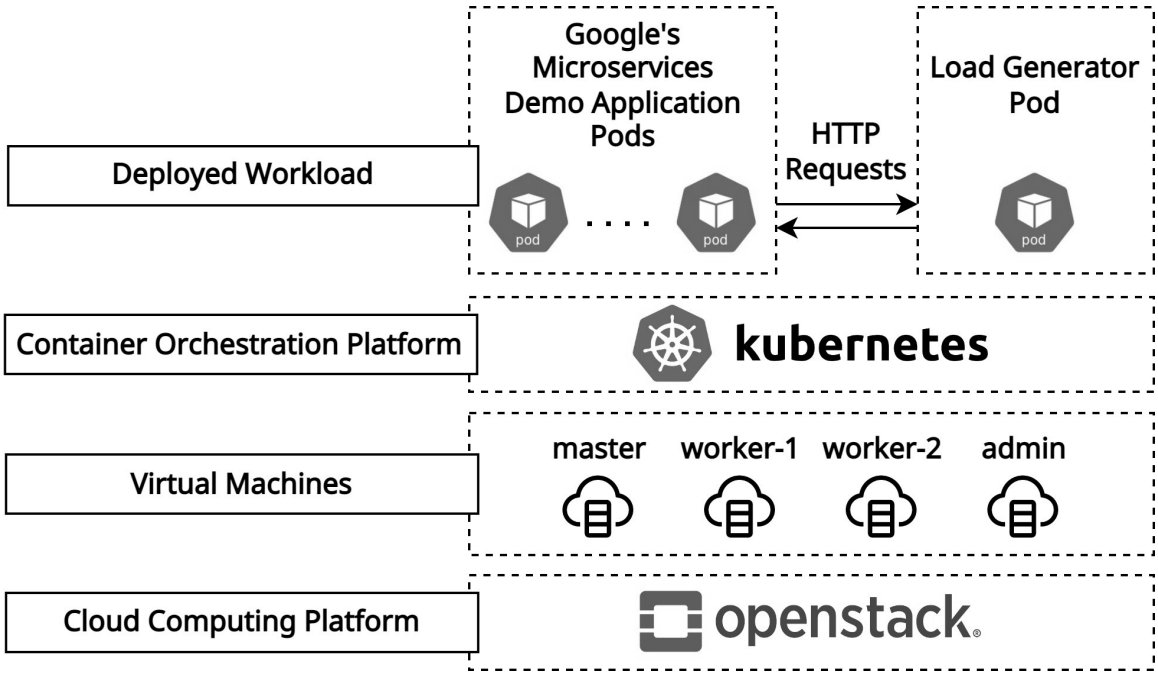


Figure 6: Experimental setup.

like deployments, nodes, and pods. We configure Prometheus to run its pods on *admin* for segregation from the application workload. Here, Prometheus serves as our cloud monitoring system. For our experiments, we use the metrics described at a high level in Table 2 and Table 3, which can reveal processor-related faults and data or concept drifts. We collect these metrics for each of the 10 pods running the deployed application microservices wherever available, summing up to 250 metrics.

Table 2: Gauge Type CPU-related cAdvisor [2] and Kube State [3] Metrics.

Metric	Description
container_spec_cpu_period	CPU period of the container
container_spec_cpu_shares	CPU share of the container
container_spec_cpu_quota	CPU quota of the container
kube_pod_container_resource_limits	The number of requested limit resource by a container
kube_pod_container_resource_requests	The number of requested request resource by a container

Table 3: Counter Type CPU-related cAdvisor Metrics [2].

Metric	Description
container_cpu_cfs_periods_total	Number of elapsed enforcement period intervals

container_cpu_cfs_throttled_ periods_total	Number of throttled period intervals
container_cpu_cfs_throttled_ seconds_total	Total time duration the container has been throttled
container_cpu_system_seconds_ total	Cumulative system CPU time consumed
container_cpu_usage_seconds_ total	Cumulative CPU time consumed
container_cpu_user_seconds_ total	Cumulative user CPU time consumed

5.1.2 Parameter Setting

We transform the metric time series to metric data instances of window size 12, as discussed in Section 4.1.3 so that each data instance carries system state information spanning 60 seconds. In our active learning module, we set T_u = 0.9 and T_l = 0.1. Consequently, $\alpha = -\{0.9 \cdot \log_2(0.9) + 0.1 \cdot \log_2(0.1)\} = 0.469$ for a desired classification confidence of more than 90% over each class. We set the parameter H used for calculating E_i^t to 6 to quantify the uncertainty in classifier predictions over the last 30 seconds. While β is calculated after each iteration of training or adaptation as the highest reconstruction error produced by the trained Encoder-Decoder network over the corresponding data set used for training or adaptation.

5.1.3 Test Suite

We define a simple test suite composed of functional and fault identification tests to identify processor-related faults that lead to non-conformance with the application specifications. For convenience, we conceptually group the application microservices into user and helper microservices. The user actions directly call upon user microservices. User microservices call upon helper microservices in turn. We define functional tests that imitate user shopping flows to check that user microservices are available and have acceptable response times. The failure of a functional test implies the failure of a user microservice or its helper microservices, as identified in the relationship matrix depicted in Table 4.

Based on functional test failures, we determine a set of possible fault-affected microservice pods P. We restrict the execution of fault identification tests to these pods. Limiting our discussion to CPU throttling issues for brevity, we define the following fault identification test to identify the underlying resource conflicts that led to the functional test failures. For each pod P_i in P, if the average CPU throttling in its server container C_i over the last 60 seconds exceeds 50%, we deduce a CPU fault in P_i. Finally, the test suite returns a data label set based on the identified pods with resource conflicts.

Table 4: Functional Test - Microservices Relationship Matrix.

Test / Microservice	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
T1	✓		✓		✓		✓	✓	✓	✓
T2	✓		✓		✓		✓	✓	✓	
T3	✓	✓	✓	✓		✓	✓		✓	✓
T4	✓		✓		✓		✓		✓	
T5	✓		✓		✓		✓		✓	
T6	✓		✓		✓		✓	✓	✓	✓

We denote functional tests by T1: Add to Cart, T2: Browse Product, T3: Checkout, T4: Home Page, T5: Set Currency, T6: View Cart. We denote microservices by M1: Cart Service, M2: Checkout Service, M3: Currency Service, M4: Email Service, M5: Product Catalog Service, M6: Payment Service, M7: Frontend, M8: Recommendation Service, M9: Redis, M10: Shipping Service

5.1.4 Programming Tools and Fault Injections

We used Python and shell scripts to implement various components of the framework. For all the machine learning operations, we used PyTorch and scikit-learn libraries. We generate fault scenarios using the stress-ng tool. We inject faults into application microservice pods by stressing the CPU from inside the pods at 50% to 100% in cumulative increments of 10%.

5.2 Results

In this section, we discuss the experimental results of our proposed framework over 11 class labels corresponding to normal system state and CPU faults in 10 microservices of the application, defined as:

- Normal System State: S_{normal}
- Cart Service CPU Fault: F_{cart_cpu}
- Checkout Service CPU Fault: $F_{checkout\_cpu}$
- Currency Service CPU Fault: $F_{currency\_cpu}$
- Email Service CPU Fault: F_{email_cpu}
- Product Catalog Service CPU Fault: $F_{product\_cpu}$
- Payment Service CPU Fault: $F_{payment\_cpu}$
- Frontend CPU Fault: $F_{frontend\_cpu}$

- Recommendation Service CPU Fault: $F_{recommendation\_cpu}$
- Redis Cache CPU Fault: F_{redis_cpu}
- Shipping Service CPU Fault: $F_{shipping\_cpu}$

For convenience, we collectively refer to the Encoder-Decoder and Encoder-Classifier networks as the classifier. We train the initial classifier over a training data set with 400 data instances corresponding to each class label exclusively, labeled manually using the fault injection mechanism. We also use this data set for hyperparameter tuning over the search space, as shown in Table 5. Table 6 summarizes the architecture of our classifier.

Table 5: Hyperparameter Search Space.

Hyperparameter	**Range**
LSTM layers in Encoder and Decoder	[1, 4]
Compressed Feature Space	[64, 128]
Optimizer	Adam, NAdam
Batch Size	[8, 32]
Learning Rate	[0.00001, 0.0005]

We consider 8 different data distribution scenarios produced sequentially to evaluate the ability of our proposed framework to identify and adapt to emerging fault patterns and data or concept drifts.

- **Scenario 1**: Simultaneous CPU faults in *frontend*, *checkout service*, and *payment service* pods.
- **Scenario 2**: Simultaneous CPU faults in *frontend*, *currency service*, *product catalog service*, and *recommendation service* pods.
- **Scenario 3**: Simultaneous CPU faults in *frontend*, *checkout service*, *payment service*, *email service*, and *shipping service* pods.
- **Scenario 4**: Simultaneous CPU faults in *cart* and *redis cache* pods.
- **Scenario 5**: Tenfold increase in user workload, and change of pod CPU request and limit configurations to induce data and concept drift.
- **Scenario 6**: Post drift CPU fault in *frontend* pod.
- **Scenario 7**: Post drift CPU fault in *currency service* pod.
- **Scenario 8**: Post drift CPU fault in *shipping service* pod.

Table 6: Encoder, Decoder, and Classifier Architecture.

Layer Type	Activation Function	Output Shape
Encoder		
Input	/	(12, 250)
LSTM 1	/	(12, 250)
LSTM 2	/	(12, 192)
LSTM 3	/	(1, 96)
FC 1	Sigmoid	(1, 96)
Decoder		
Input	/	(1, 96)
Repeat	/	(12, 96)
LSTM 1	/	(12, 192)
LSTM 2	/	(12, 250)
LSTM 3	/	(12, 250)
Flatten	/	(1, 3000)
FC 1	Sigmoid	(1, 3000)
View	/	(12, 250)
Classifier		
Input	/	(1, 96)
FC 1	ReLU	(1, 48)
FC 2	Sigmoid	(1, 11)

Here, scenarios 1 to 4 are each followed by the normal system state. Post scenario 5, this data or concept drifted distribution becomes the new normal system state. Scenarios 6 to 7 are each followed by the new normal system state.

When the framework identifies an emerging data distribution with the assistance of the informativeness estimators, it triggers the test suite to label these data instances. We configure the framework to begin the classifier adaptation process on acquiring 400 labeled data instances from the emerging data distribution. The existing data set is then augmented with the newly acquired labeled data instances. Subsequently, we adapt the classifier over the augmented data set using the transfer learning mechanism. Table 7 shows the training times for the initial classifier and its subsequent adaptations over each scenario, and the time elapsed before triggering the test suite based on the corresponding informativeness estimators for each scenario.

While the training time for the initial classifier is 708.8s, the training times for its subsequent adaptations range from 321.3s to 541.4s, averaging 438.6s. Even though the size of the training data set increases by 400 data instances in each iteration of classifier adaptation, we achieve considerably lower training times for classifier adaptations with the aid of the transfer learning mechanism, averaging 38.1% lower, as compared to the training time for the initial classifier. The time elapsed before triggering the test suite for various

Table 7: Training Time and Time Delay Before Triggering Test Suite.

Scenario	Training Time (s)	Detection Time (s)
Initial Training	708.8	-
Scenario 1	324.8	45
Scenario 2	321.3	30
Scenario 3	387.0	35
Scenario 4	402.1	30
Scenario 5	541.4	5
Scenario 6	521.8	30
Scenario 7	500.3	40
Scenario 8	509.9	80

Models are trained on a physical machine with Intel(R) Core(TM) i5-9300H 2.40GHz CPU and 8 GB RAM

emerging data distribution scenarios ranges from 5s to 80s, averaging 37s, where metrics are collected from the managed cloud system every 5 seconds. The concept and data drift induced by Scenario 5 introduces drastic changes in the data distribution, which is identified very quickly in 5s by the proposed framework. On the other hand, the classifier in the proposed framework generalizes well and achieves high classification confidence by the time of induction of Scenario 8. Thus, it takes a relatively long time of 80s before triggering the test suite for Scenario 8.

Given that none of the existing data labeling techniques are suitable for our purpose as discussed in Section 3.3, we use the initial classifier's performance as the baseline to measure the proposed framework's efficacy in the presence of emerging data distributions. We use a validation data set for evaluations consisting of data instances from each emerging data distribution scenario for validation over emerging data distributions. Additionally, it consists of data instances corresponding to each fault class label exclusively and the normal system state obtained before introducing emerging data distribution scenarios for validation over known data distributions. To account for class imbalances in the validation data set, we use weighted F1-score as the performance metric. Figure 7 shows the obtained weighted F1-score vs. timestep for our eight scenarios of emerging data distributions described above. As apparent, without adaptation, the initial F1-score of 0.99 over known data distributions begins to drop as we encounter emerging data distributions starting from Scenario 1 and continues to drop to 0.70 at the end of Scenario 8. However, by identifying and adapting to emerging data distributions with the proposed framework over eight iterations, we achieve a consistent F1-score of 0.99, which is 41% higher.

The initial classifier, trained on a data set that only consists of data instances corresponding to the presence of a single fault in the system at any point in time, cannot correctly identify the presence of multiple simultaneous faults in the system, as produced by Scenarios

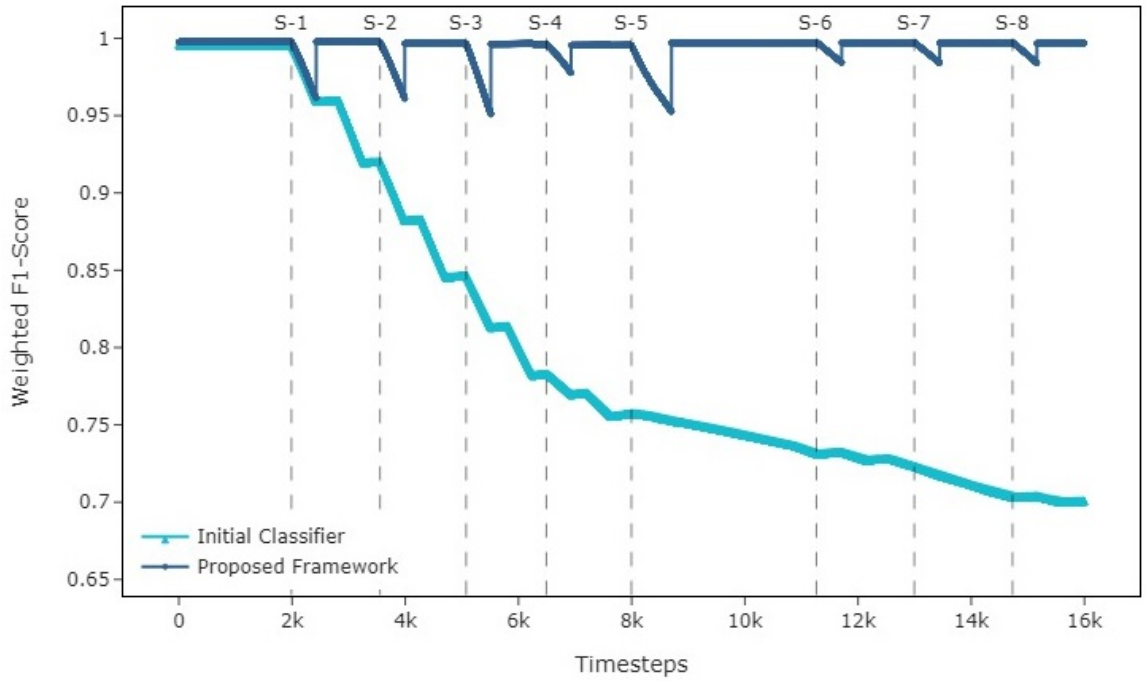
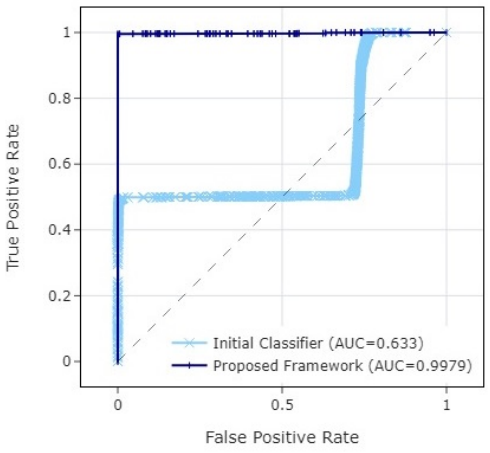
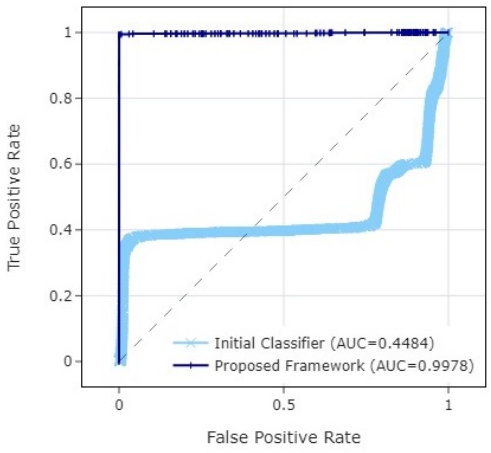


Figure 7: Weighted F1-score vs. time over emerging data distribution scenarios marked as S-1 to S-8.

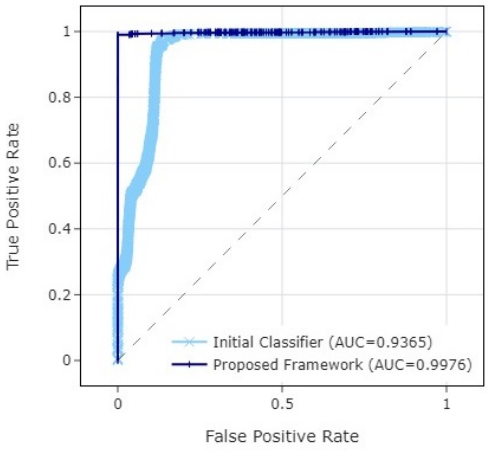
1 to 4. To begin with, it is not feasible to obtain a training data set comprising data instances corresponding to all the possible combinations of multiple simultaneous faults in the system. However, the proposed framework can identify such data instances as and when there are multiple simultaneous faults in the system and exploit them to improve the classifier through adaptation. This mechanism enables the classifier to create better decision boundaries to isolate such faults and label data points with multiple fault class labels. Furthermore, in conventional systems, the induction of concept and data drifts, as produced by Scenario 5, would necessitate reacquiring a new training data set representative of all class labels to retrain the classifier. However, the proposed framework can adapt to concept and data drifts by collecting data instances corresponding to various class labels from emerging data distributions as and when they occur. We would expect similar drops in F1-score at the induction of post-drift faults produced by Scenarios 6, 7, and 8 as the induction of data and concept drift produced by Scenarios 5. However, we observed relatively lesser degradation in F1-score for Scenarios 6, 7, and 8 than Scenario 5, which suggests that the classifier attains better generalization capabilities by learning from a wider variety of data over several iterations of adaptations. At any point in time during the evaluation, our validation data set comprises data instances from all previously seen data distributions from before or after the introduction of emerging data distribution scenarios in the system. We can observe that the proposed framework does not override previously learned patterns while learning new patterns in the data. Moreover, it incorporates all seen data distribution patterns to extract better latent feature representations and achieve better generalization.



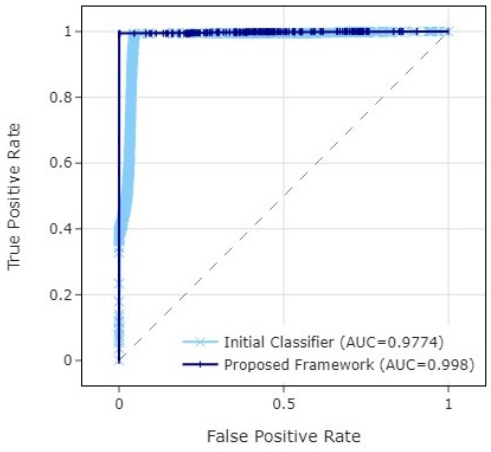
(a) F_{cart_cpu}



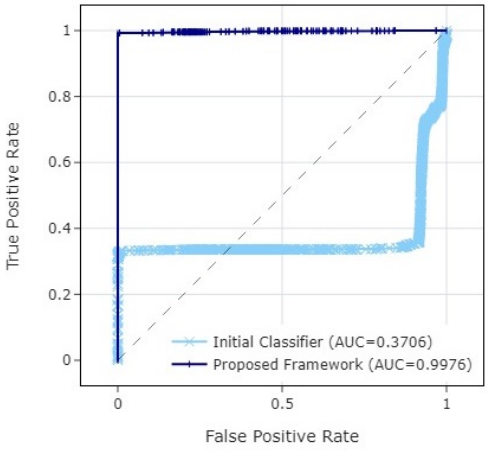
(b) $F_{checkout\_cpu}$



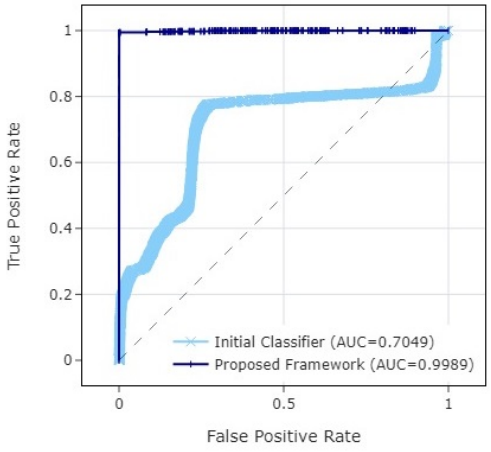
(c) $F_{currency\_cpu}$



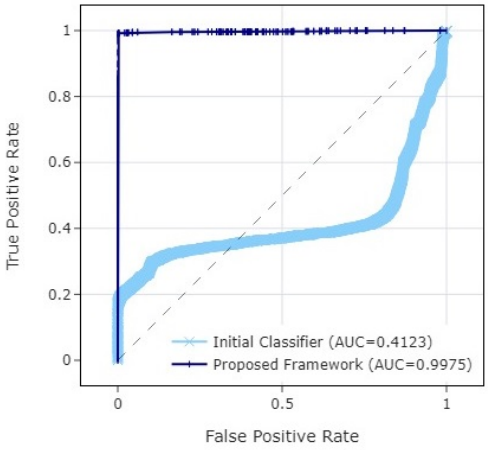
(d) F_{email_cpu}



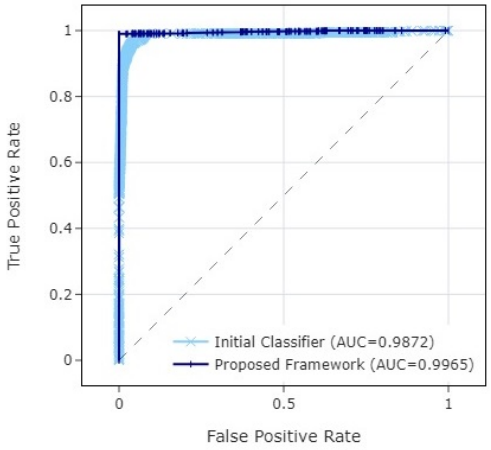
(e) $F_{product\_cpu}$



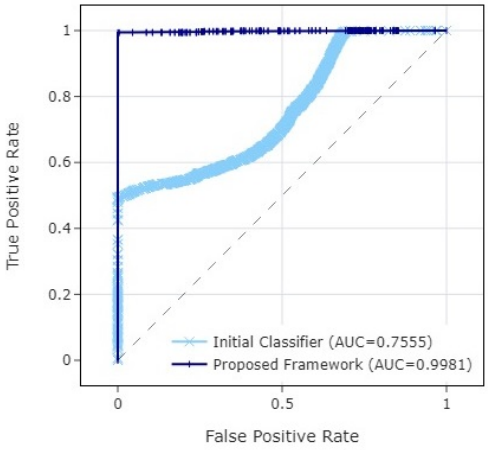
(f) $F_{payment\_cpu}$



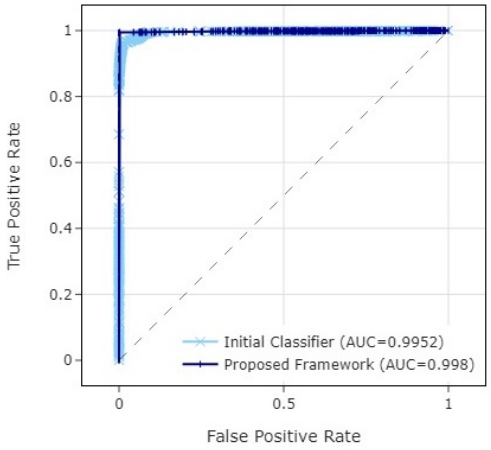
(g) $F_{frontend\_cpu}$



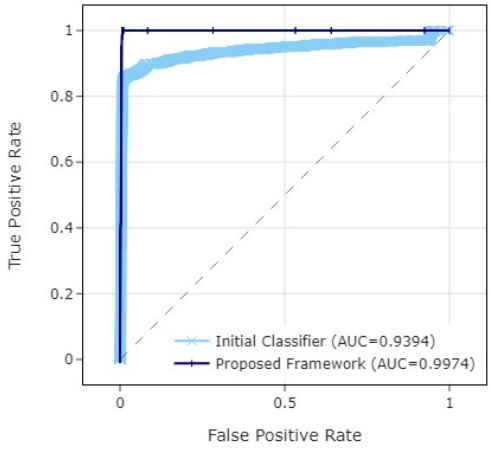
(h) $F_{recommendation\_cpu}$



(i) F_{redis_cpu}



(j) $F_{shipping\_cpu}$



(k) S_{normal}

Figure 8: Comparison of ROC curves and AUC scores with and without adaptation.

Again using the initial classifier's performance as the baseline, we evaluate the proposed framework's efficacy over each class in the presence of emerging data distributions as produced by the eight scenarios described above. As the classifier produces probabilistic scores over each class, the final classifications depend on the probability threshold configurations. To present an evaluation independent of these configurations, we use One-vs-Rest Receiver Operating Characteristic (ROC) curves and Area Under the ROC Curve (AUC) scores which take all classification thresholds into account. Figure 8 shows a comparison of OvR ROC curves and AUC scores with and without adaptation using the proposed framework over the validation data set for each of the 11 classes. In the presence of emerging data distributions, the AUC scores degrade over most of the 11 classes, averaging 0.7419. However, with adaptation using the proposed framework, we achieve AUC scores of around 0.99 over all the classes, averaging 0.9978, which is 34% higher.

Though the initial classifier achieves satisfactory results for F_{email_cpu}, $F_{recommendation\_cpu}$, and $F_{shipping\_cpu}$ classes, the proposed framework achieves significant improvements for F_{cart_cpu}, $F_{checkout\_cpu}$, $F_{currency\_cpu}$, $F_{product\_cpu}$, $F_{payment\_cpu}$, $F_{frontend\_cpu}$, F_{redis_cpu}, and S_{normal} classes as compared to the initial classifier. For one, we can attribute the degradation of the performance of the initial classifier to the absence of data instances corresponding to multiple simultaneous faults in the system in the training data set, which leads to subpar decision boundaries. The improvements achieved by the proposed framework can be credited to its capability to continuously discover informative data instances, including those corresponding to multiple simultaneous faults in the system, and exploit them to improve the classifier through adaptation to create better decision boundaries for isolating multiple simultaneous faults in the system with high efficacy.

The experiments demonstrate that the proposed framework efficiently labels time series data from a high-dimensional space of 250 cloud metrics over a time window size of 60 seconds, satisfying Requirements 1 and 4. It performs multi-label classification over 11 classes with high F1 and AUC scores, in consonance with Requirement 3. As evident from our experiments over 8 emerging data distribution scenarios, the proposed framework can discover and assimilate emerging fault patterns and concept or data drifts as specified in Requirement 5. We eliminate the human intervention required in conventional active learning settings by replacing the crowd component with a test suite in the proposed framework. Furthermore, the proposed framework can be adjusted to perform data labeling at various granularities by defining appropriate test suites to assign data labels according to the granularity requirements. Thus, it meets Requirements 2 and 6.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In recent years, there has been an upsurge in the use of cloud-based services and products, migration from legacy enterprise software to cloud-based tools, and migration of workloads from on-premise infrastructure to the cloud. While cloud computing offers a wide range of benefits, cloud environments are inevitably prone to faults, which may lead to service level agreement breaches and even downtimes, causing enormous economic losses. Thus, it is of utmost importance to promptly detect and mitigate these faults to ensure that the applications deployed in cloud environments work smoothly. While we can find several machine learning-based solutions for fault detection in cloud environments in the literature, the data labeling aspects of the problem have not received due attention. To elucidate the data labeling problem for fault detection in cloud environments, we presented an illustrative scenario resembling real-world settings, comprising a microservice-based application deployed in a cloud environment. Using this illustrative scenario, we derived a set of requirements essential to a befitting data labeling solution. Most importantly, we established that it should be able to identify and adapt to emerging data distributions and eliminate human intervention from the labeling process. While evaluating the literature with respect to these requirements, we found that the existing data labeling techniques do not meet these requirements satisfactorily. Moreover, we observed that most machine learning-based solutions for fault detection in cloud environments rely on the fault injection mechanism to acquire the labeled data for training or evaluation, which has several inherent limitations. To cope with the aforementioned challenges, in this thesis we proposed a framework for automated labeling of cloud metric data instances with the corresponding cloud system state, which meets all the derived requirements. Our framework adopts the concept of active learning to identify informative data instances. By delegating these informative data instances to a test suite for labeling in real-time, our framework replaces the crowd component in conventional active learning settings to eliminate human intervention and achieve automation. Moreover, our framework can maintain persistent labeling accuracy by actively identifying and adapting to emerging data distributions. To validate the efficacy of our framework, we presented

a proof of concept on a real-world testbed comprising a microservice-based application deployed in a cloud environment. Our results over eight scenarios of emerging data distributions demonstrate that our framework achieves significant performance improvements over a baseline classifier by maintaining persistent F1 and AUC scores.

6.2 Future Work

While we have evaluated the proposed framework for a specific use-case with processor-related faults, it can be extended to label a wider variety of faults by incorporating relevant cloud metrics in the active learning classifier and devising pertinent test suites. Moreover, it can be extended to label metric data exposed by any system by devising an appropriate test suite according to the desired concept of interest for identifying the corresponding operational status of the system in real time. Furthermore, we can investigate the integration of existing test suites such as OpenStack Tempest [56] or Kubernetes E2E Tests [55] designed for system testing with the framework to label metric data exposed by OpenStack or Kubernetes systems with their underlying faults, respectively. Another interesting research direction would be to investigate the utilization of the classifier predictions for test case selection or prioritization to further optimize the use of the framework. Additionally, as the acquired labeled data set grows over time, a sampling mechanism may be needed to frame an appropriate adaptation training data set while accounting for class imbalances.

Chapter 7

Publications

- P. Bagora, A. Ebrahimzadeh, F. Wuhib, R. H. Glitho, "Data Labeling for Fault Detection in Cloud: A Test Suite-Based Active Learning Approach," in *Proc. IEEE International Conference on Network Softwarization (NetSoft)*, 2023, to appear
- P. Bagora, A. Ebrahimzadeh, F. Wuhib, R. H. Glitho, "Method and System for Data Labeling in a Cloud System through Machine Learning," *Provisional US Patent Application Number: 63/426584* (Full Patent Application Filed)

References

[1] "Google's Online Boutique Microservices Demo Application." [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[2] "Monitoring cAdvisor with Prometheus." [Online]. Available: https://github.com/google/cadvisor/blob/master/docs/storage/prometheus.md

[3] "Kube State Metrics: Pod Metrics." [Online]. Available: https://github.com/kubernetes/kube-state-metrics/blob/main/docs/pod-metrics.md

[4] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *SIGCOMM Computer Communication Review*, vol. 39, no. 1, p. 50–55, Dec 2009.

[5] A. U. Rehman, R. L. Aguiar, and J. P. Barraca, "Fault-tolerance in the scope of cloud computing," *IEEE Access*, vol. 10, pp. 63 422–63 441, 2022.

[6] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017.

[7] A. Bhatia, M. Gerndt, and J. Cardoso, "Efficient failure diagnosis of openstack using tempest," *IEEE Internet Computing*, vol. 22, no. 6, pp. 61–70, Nov-Dec 2018.

[8] B. Settles, "Automating inquiry," in *Active Learning*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012, vol. 6, no. 1.

[9] J. Wang and Y. Chen, "Introduction," in *Introduction to Transfer Learning*, ser. Machine Learning: Foundations, Methodologies, and Applications. Springer International Publishing, 2018.

[10] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[11] M. Soualhia, C. Fu, and F. Khomh, "Infrastructure fault detection and prediction in edge cloud environments," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, p. 222–235.

[12] K. T. Bui, L. Van Vo, C. M. Nguyen, T. V. Pham, and H. C. Tran, "A fault detection and diagnosis approach for multi-tier application in cloud computing," *Journal of Communications and Networks*, vol. 22, no. 5, pp. 399–414, 2020.

[13] D. D. Lewis and W. A. Gale, "A sequential algorithm for training text classifiers," in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1994, pp. 3–12.

[14] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, H. R. Motahari-Nezhad, E. Bertino, and S. Dustdar, "Quality control in crowdsourcing systems: Issues and directions," *IEEE Internet Computing*, vol. 17, no. 2, pp. 76–81, 2013.

[15] D. Yarowsky, "Unsupervised word sense disambiguation rivaling supervised methods," in *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics*, 1995, p. 189–196.

[16] X. Zhu, Z. Ghahramani, and J. Lafferty, "Semi-supervised learning using gaussian fields and harmonic functions," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, 2003, p. 912–919.

[17] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, "Snorkel: Rapid training data creation with weak supervision," *Proceedings of the VLDB Endowment*, vol. 11, no. 3, p. 269–282, Nov 2017.

[18] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Tech. Rep. 800-145, Sep 2011.

[19] R. Buyya, J. Broberg, and A. M. Goscinski, "Introduction to cloud computing," in *Cloud Computing : Principles and Paradigms*. John Wiley & Sons, 2011.

[20] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2021.

[21] H. Agarwal and A. Sharma, "A comprehensive survey of fault tolerance techniques in cloud computing," in *International Conference on Computing and Network Communications*, 2015, pp. 408–413.

[22] M. K. Gokhroo, M. C. Govil, and E. S. Pilli, "Detecting and mitigating faults in cloud computing environment," in *3rd International Conference on Computational Intelligence & Communication Technology*, 2017, pp. 1–9.

[23] "Prometheus Alerting." [Online]. Available: https://prometheus.io/docs/alerting/latest/overview/

[24] A. Cauveri and R. Kalpana, "Dynamic fault diagnosis framework for virtual machine rolling upgrade operation in google cloud platform," in *2017 International Conference on Power and Embedded Drive Control (ICPEDC)*, 2017, pp. 235–241.

[25] M. Smara, M. Aliouat, A.-S. K. Pathan, and Z. Aliouat, "Acceptance test for fault detection in component-based cloud computing and systems," *Future Generation Computer Systems*, vol. 70, pp. 74–93, 2017.

[26] E. Kinsbruner, "Different Types of Testing in Software." [Online]. Available: https://www.perfecto.io/resources/types-of-testing

[27] "Test Suite Tutorial: Comprehensive Guide with Best Practices." [Online]. Available: https://www.lambdatest.com/learning-hub/test-suite

[28] R. Saravanan and P. Sujatha, "A state of art techniques on machine learning algorithms: A perspective of supervised learning approaches in data classification," in *Second International Conference on Intelligent Computing and Control Systems*, 2018, pp. 945–949.

[29] O. Simeone, "A very brief introduction to machine learning with applications to communication systems," *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 4, pp. 648–664, 2018.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, May 2015.

[31] S. Haykin, "Introduction," in *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1998.

[32] C. Aggarwal, X. Kong, Q. Gu, J. Han, and P. Yu, "Active learning: A survey," in *Data Classification*. CRC Press, Jan. 2014, pp. 571–605.

[33] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural Computation*, vol. 4, no. 1, pp. 1–58, 1992.

[34] W. Dai, Q. Yang, G.-R. Xue, and Y. Yu, "Boosting for transfer learning," in *Proceedings of the 24th International Conference on Machine Learning*, 2007, p. 193–200.

[35] S. Gao, H. Luo, D. Chen, S. Li, P. Gallinari, and J. Guo, "Cross-domain recommendation via cluster-level latent factor model," in *Machine Learning and Knowledge Discovery in Databases*, 2013, pp. 161–176.

[36] Y. Roh, G. Heo, and S. E. Whang, "A survey on data collection for machine learning: A big data - ai integration perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1328–1347, 2021.

[37] "About ImageNet: Summary and Statistics." [Online]. Available: https://web.archive.org/web/20200907212153/http://image-net.org/about-stats.php

[38] S. Newman, "Microservices," in *Building Microservices*. O'Reilly Media, 2015.

[39] S. Gupta, N. Muthiyan, S. Kumar, A. Nigam, and D. A. Dinesh, "A supervised deep learning framework for proactive anomaly detection in cloud workloads," in *14th IEEE India Council International Conference*, 2017, pp. 1–6.

[40] B. Shayesteh, C. Fu, A. Ebrahimzadeh, and R. Glitho, "Auto-adaptive fault prediction system for edge cloud environments in the presence of concept drift," in *IEEE International Conference on Cloud Engineering*, 2021, pp. 217–223.

[41] R. A. Ghalehtaki, A. Ebrahimzadeh, F. Wuhib, and R. H. Glitho, "An unsupervised machine learning-based method for detection and explanation of anomalies in cloud environments," in *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2022, pp. 24–31.

[42] J. Li, R. Huang, G. He, S. Wang, G. Li, and W. Li, "A deep adversarial transfer learning network for machinery emerging fault detection," *IEEE Sensors Journal*, vol. 20, no. 15, pp. 8413–8422, 2020.

[43] L. Wen, L. Gao, and X. Li, "A new deep transfer learning based on sparse auto-encoder for fault diagnosis," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 1, pp. 136–144, 2019.

[44] Z.-H. Zhou and M. Li, "Tri-training: exploiting unlabeled data using three classifiers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1529–1541, 2005.

[45] Y. Zhou and S. Goldman, "Democratic co-learning," in *16th IEEE International Conference on Tools with Artificial Intelligence*, 2004, pp. 594–602.

[46] A. Blum and T. Mitchell, "Combining labeled and unlabeled data with co-training," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 1998, p. 92–100.

[47] A. Blum and S. Chawla, "Learning from labeled and unlabeled data using graph mincuts," in *Proceedings of the Eighteenth International Conference on Machine Learning*, 2001, p. 19–26.

[48] P. Talukdar and W. Cohen, "Scaling Graph-based Semi Supervised Learning to Large Number of Labels Using Count-Min Sketch," in *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, vol. 33, Apr 2014, pp. 940–947.

[49] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by committee," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, 1992, p. 287–294.

[50] G. Li, J. Wang, Y. Zheng, and M. J. Franklin, "Crowdsourced data management: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2296–2319, 2016.

[51] F. Daniel, P. Kucherbaev, C. Cappiello, B. Benatallah, and M. Allahbakhsh, "Quality control in crowdsourcing: A survey of quality attributes, assessment techniques, and assurance actions," *ACM Computing Surveys*, vol. 51, no. 1, Jan 2018.

[52] J. C. Chang, S. Amershi, and E. Kamar, "Revolt: Collaborative crowdsourcing for labeling machine learning datasets," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, p. 2334–2346.

[53] V. Crescenzi, P. Merialdo, and D. Qiu, "Crowdsourcing large scale wrapper inference," *Distributed and Parallel Databases*, vol. 33, no. 1, p. 95–122, Mar 2015.

[54] M. Schaekermann, J. Goh, K. Larson, and E. Law, "Resolvable vs. irresolvable disagreement: A study on worker deliberation in crowd work," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, Nov 2018.

[55] "End-to-End Testing in Kubernetes." [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-testing/e2e-tests.md

[56] "Tempest Testing Project." [Online]. Available: https://docs.openstack.org/tempest/latest/