Predicting Performance Anomalies in Software Systems at Run-time

GUOLIANG ZHAO, School of Computing, Queen's University, Canada
SAFWAT HASSAN, Department of Engineering, Thompson Rivers University, Canada
YING ZOU, Department of Electrical and Computer Engineering, Canada
DEREK TRUONG, IBM, Canada
TOBY CORBIN, IBM, United Kingdom

High performance is a critical factor to achieve and maintain the success of a software system. Performance anomalies represent the performance degradation issues (e.g., slowing down in system response times) of software systems at run-time. Performance anomalies can cause a dramatically negative impact on users' satisfaction. Prior studies propose different approaches to detect anomalies by analyzing execution logs and resource utilization metrics after the anomalies have happened. However, the prior detection approaches cannot predict the anomalies ahead of time; such limitation causes an inevitable delay in taking corrective actions to prevent performance anomalies from happening. We propose an approach that can *predict performance anomalies* in software systems and raise anomaly warnings in advance. Our approach uses a Long-Short Term Memory neural network to capture the normal behaviors of a software system. Then, our approach predicts performance anomalies by identifying the early deviations from the captured normal system behaviors. We conduct extensive experiments to evaluate our approach using two real-world software systems (i.e., Elasticsearch and Hadoop). We compare the performance of our approach with two baselines. The first baseline is one state-to-the-art baseline called Unsupervised Behavior Learning. The second baseline predicts performance anomalies by checking if the resource utilization exceeds pre-defined thresholds. Our results show that our approach can predict various performance anomalies with high precision (i.e., 97–100%) and recall (i.e., 80–100%), while the baselines achieve 25–97% precision and 93–100% recall. For a range of performance anomalies, our approach can achieve sufficient lead times that vary from 20 to 1,403 s (i.e., 23.4 min). We also demonstrate the ability of our approach to predict the performance anomalies that are caused by real-world performance bugs. For predicting performance anomalies that are caused by real-world performance bugs, our approach achieves 95–100% precision and 87–100% recall, while the baselines achieve 49–83% precision and 100% recall. The obtained results show that our approach outperforms the existing anomaly prediction approaches and is able to predict performance anomalies in real-world systems.

CCS Concepts: • **Software and its engineering → Software post-development issues**;

Additional Key Words and Phrases: Performance anomaly prediction, LSTM neural network, software systems

Authors' addresses: G. Zhao, School of Computing, Queen's University, Kingston, Canada; email: 17gz2@cs.queensu.ca; S. Hassan, Department of Engineering, Thompson Rivers University, Kamloops, Canada; email: shassan@tru.ca; Y. Zou, Department of Electrical and Computer Engineering, Kingston, Canada; email: ying.zou@queensu.ca; D. Truong, IBM, Markham, Canada; email: trong@ca.ibm.com; T. Corbin, IBM, Southampton, United Kingdom; email: corbint@uk.ibm.com.

1049-331X/2021/04-ART33 $15.00
https://doi.org/10.1145/3440757

ACM Reference format:
Guoliang Zhao, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. 2021. Predicting Performance Anomalies in Software Systems at Run-time. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 33 (April 2021), 33 pages.
https://doi.org/10.1145/3440757

1 INTRODUCTION

Large-scale software systems have become an essential part of our daily activities, such as performing financial services, fulfilling healthcare operations, and enabling interpersonal communication. To meet users' expectations, software systems need to perform their functionalities with high performance and reliability. However, the performance of software systems is prone to run-time performance anomalies (e.g., slowing down in system response time) [55]. Performance anomalies mean that the monitored system behaviors cannot be explained by the current system workload [9]. For example, the number of transactions that are processed by a system suggests less CPU and memory consumption than the actual resource usage.

To maintain the performance of software systems, operators need to detect performance anomalies and prevent failures from happening at run-time [29]. Due to the increasing scale and complexity of software systems, it is challenging for operators to manually keep track of the execution status of software systems. Hence, researchers have proposed various approaches to automatically monitor software systems and detect anomalies at run-time [27, 29, 44, 45, 51, 64].

However, the delay in detecting anomalies and taking corrective actions can result in the violations of Service Level Objective (e.g., long response time for users' requests) or system failures, which can cause financial loss [11]. For example, in August 2013, Amazon was down for 40 min. Due to this outage, Amazon had almost $5 million revenue loss [61]. Hence, it is desirable to provide a performance anomaly prediction approach that can proactively raise anomaly warnings in advance, and thereby help operators to prevent potential anomalies from happening.

Performance anomaly prediction approaches forecast that a software system is about to enter an anomalous state by capturing the pre-failure state before the anomaly happens [11]. Performance bugs represent the non-functional bugs, which can cause significant performance degradation [30]. Performance bugs (e.g., memory leak bugs) do not always break down software systems instantly. Software systems are considered to be in a pre-failure state from the start of experiencing performance degradation until the performance anomaly occurs [11]. Prior studies propose approaches that can predict failures of virtual machines (VMs) in cloud infrastructures [11, 56]. The existing studies [11, 56] analyze the metrics collected from VMs (e.g., the CPU usage of a VM) and predict the future failures of the monitored VMs. For a cloud with hundreds of VMs, predicting VM-level failures can pinpoint the VM that will have a failure and enable operators to take proper actions to prevent the failure from happening. For example, if a VM in the cloud is running out of memory, to stop the out of memory failure from happening, operators can migrate the VM to a physical host with larger memory space [57]. However, the existing studies [11, 56] predict failures at the VM-level, and operators still need to manually localize the anomalous application that causes the predicted failure. There are multiple challenges for achieving an efficient performance anomaly prediction for applications (i.e., achieving performance anomaly prediction at the *application-level*) as follows.

- **Dynamic software behaviors**. The behaviors of software systems are dynamic (e.g., fluctuating memory usage) because of the constantly changing user behaviors and varying system workloads. Hence, it is difficult to characterize the behaviors of a software system and precisely capture the pre-failure state [26]. Consequently, existing anomaly detection

techniques that use statistical models can easily misclassify the temporal high resource utilization as anomalous behaviors [7, 41, 42, 43, 59].

- **Instrumentation challenges**. Operators do not always have access to the source code of systems running on the cloud. It is infeasible to apply existing anomaly detection techniques [4, 17] that need to instrument the source code of software systems. In addition, instrumenting the binary or source code of software systems can introduce overhead and impact the performance of the systems at run-time [17, 60].
- **Data acquisition challenges**. It is challenging to obtain enough labeled training data (i.e., monitoring data with normal and anomalous labels) to build supervised anomaly prediction approaches [19, 24, 66]. Companies are often reluctant to release their monitoring dataset as it may contain confidential information about their users. More importantly, supervised techniques can only predict known anomalies that appear in the training dataset [11].

In this article, we present a *performance anomaly prediction* approach that overcomes the aforementioned challenges. First, we apply a Long-Short Term Memory (LSTM) neural network to handle the difficulty in characterizing the dynamic behaviors of applications. LSTM neural network overcomes the vanishing gradient problem experienced by Recurrent Neural Network (RNN) [23]. LSTM neural network has been widely used in language modeling [54], text classification [71], and failure prediction for physical equipment (e.g., engines [7, 41, 42, 43, 59]. Compared with other machine learning techniques (e.g., linear regression) and deep learning techniques (e.g., RNN), LSTM neural network has the advantage to incorporate the time dependency in time-series data. By incorporating the time dependency, the LSTM neural networks can capture both the temporal fluctuations and the long-term changes in the monitoring data (e.g., CPU usage) collected from an application to characterize the dynamic behaviors of the application.

To avoid the instrumentation of applications, we only use the performance monitoring application that is external to the applications to track resource utilization (e.g., the CPU and memory usage of a JAVA application). To solve the challenges related to data acquisition, we train LSTM neural networks to capture the normal behaviors of applications under regular operations. Compared with obtaining labeled data from normal and anomalous behaviors of applications, collecting only normal behaviors data is more straightforward and practical [11]. At the run-time monitoring stage, our approach predicts anomalies by checking whether the application deviates from the normal behaviors that are expected from the LSTM neural networks. Therefore, our approach apply the LSTM neural networks to predict performance anomalies without the need to train our approach using anomalous behaviors data.

Predicting performance anomalies at the application-level enables our approach to be used in multiple scenarios. Applications can be hosted on a large-scale cluster with thousands of nodes (e.g., virtual machines). Our approach can be used to monitor the instances of an application running on each node and locate the anomalous instances of the application in the cluster. In addition, in an industrial environment, there might be several applications running on the same machine or VM. Operators can deploy multiple instances of our approach with each instance monitoring an application. By monitoring each application, our approach can predict which application is the anomalous application running on a machine.

The lead times measure the amount of time that our approach can predict performance anomalies in advance. After predicting a performance anomaly, operators or automatic anomaly prevention approaches [57] should take actions to prevent the predicted anomaly from happening. Anomaly prevention actions, such as scaling VM resources and live VM migration, take time to complete. Thus, our approach should predict anomalies with sufficient lead times to ensure that the anomaly prevention actions can be finished.

We conduct quantitative experiments on two different widely used open-source software systems: (1) Elatsticsearch, an open-source, distributed, and RESTful search engine [15], and (2) Hadoop MapReduce sample applications provided by Hadoop distribution [3]. Especially, our work addresses the following research questions (RQs):

RQ1. What is the performance of our approach for predicting performance anomalies?

To evaluate the performance of our approach, we inject bugs that can cause performance anomalies into the source code of the studied systems and trigger the injected performance bugs. Furthermore, we compare our approach with two baselines: (1) we implement the Unsupervised Behavior Learning (UBL) baseline proposed by Dean et al. [11]. (2) we implement a baseline that predicts anomalies for an application by checking if the resource utilization of the application exceeds pre-defined thresholds. Our experiment results demonstrate that our approach can predict various performance anomalies with 97–100% precision and 80–100% recall, while the two baselines achieve 25–97% precision and 93–100% recall.

RQ2. How early our approach can raise a performance anomaly warning before the anomaly occurs?

In this RQ, we evaluate the lead times of our approach in predicting performance anomalies. Through our experiments, our approach can predict different types of performance anomalies in advance and achieve lead times varying from 20 to 1,403 s (i.e., 23.4 min). As mentioned in the existing study [57], anomaly prevention actions, such as scaling VM resources, can be completed within 1 s. Thus, the lead times achieved by our approach can be sufficient for the automatic prevention approaches [57] to take proper actions and prevent the anomalies from happening.

RQ3. Could our approach be used to predict anomalies happened in the real world?

In the previous RQs, we measure the performance of our approach in predicting performance anomalies that are caused by the injected bugs. However, the real-world performance bugs may be more complex than our injected bugs. Hence, in this RQ, we demonstrate the ability of our approach for predicting performance anomalies that are caused by real-world performance bugs. Because of the limited and ambiguous information in bug reports, reproducing real-world performance bugs is a challenging and time-consuming task [50]. In addition, there are performance bugs that can only occur under special workloads and running environments. To the end, we manage to reproduce five performance bugs that appeared in Elasticsearch. Then, we use the reproduced performance bugs to trigger performance anomalies. We observe that our approach can predict the anomalies that are caused by these five real-world performance bugs with 95-100% precision and 87–100% recall, while the two baselines achieve 49–83% precision and 100% recall. In addition, we evaluate the run-time overhead of our anomaly prediction approach. We observe that our LSTM neural networks take an average of 0.25 ms, 320 MB memory, and 25% CPU usage (i.e., two CPU cores) on our machine with an 8-core Intel i7-4790 3.60 GHz CPU to predict anomalies at every second.

Article organization: The rest of the article is organized as follows. Section 2 describes the design of our approach. Section 3 outlines the data collection methodology. Section 4 presents the evaluation results for our approach. We discuss the overhead and usage scenarios of our approach in Section 5. The threats to validate are discussed in Section 6. We present the related work in Section 7. Finally, we conclude our article in Section 8.

2 SYSTEM DESIGN

Figure 1 gives an overview of our approach. As shown in Figure 1, our approach consists of two phases: (1) the *offline training phase* to train our LSTM neural network, and (2) the *run-time monitoring phase* to predict performance anomalies. In this section, we introduce the design of our performance anomaly prediction approach. We first describe the metrics that are used to monitor

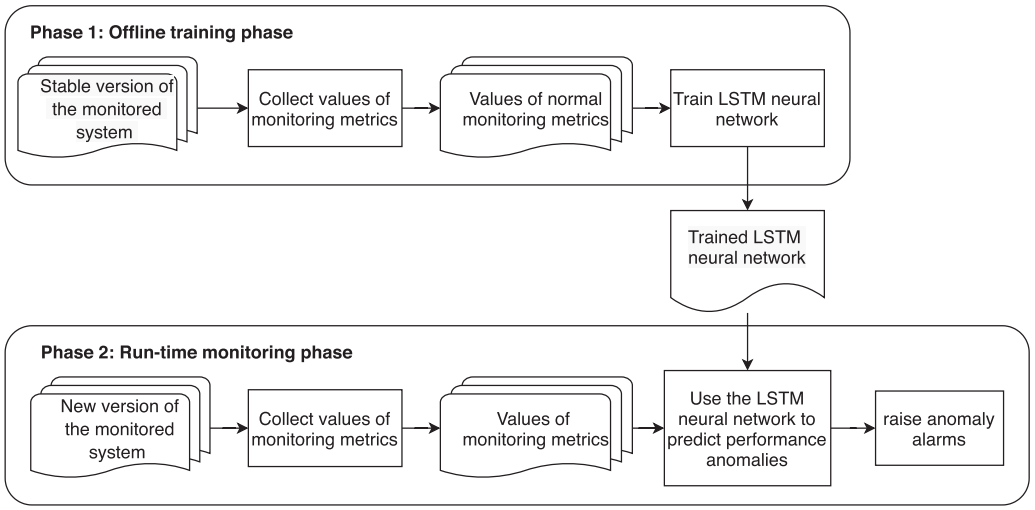


Fig. 1. An overview of our performance anomaly prediction approach.

Table 1. The Metrics Used to Monitor Applications in Our Approach

Metric Name	Description
CPUUsage	The CPU usage of the monitored application
HeapUsage	The size of the heap of the monitored application
NativeUsage	The size of the native memory space of the monitored application
EdenUsage	The size of the eden region in the heap
SurUsage	The size of the survivor region in the heap
OldGenUsage	The size of the old generation region in the heap
CodeCacheUsage	The size of the code cache region in the heap
MetaUsage	The size of the meta region in the native memory space
CompressedClassUsage	The size of the compressed class region in the native memory space
NumThreads	The number of live threads that are issued by the monitored application
NumClasses	The number of classes of the monitored application that are loaded into memory
GcTime	The time spent in the latest garbage collection (in milliseconds)

applications. Next, we introduce the architecture of our approach using the LSTM neural network. Then, we present the *training* phase of approach. Finally, we describe our approach in *predicting* performance anomaly using the trained LSTM neural network.

2.1 Monitoring Applications

In a real-world system, an application might co-exist with other applications. We measure the resource utilization that is specific to the application under monitoring using performance monitoring tools. Various performance monitoring tools (e.g., jconsole [47], Solarwinds [52], and tasks manager [65]) have been developed to monitor the resource utilization of an application. The performance monitoring tools can capture the resource utilization of the running processes of an application without instrumenting the application.

Table 1 represents the monitoring metrics that are used in our approach. We apply six metrics (e.g., CPUUsage, HeapUsage, and NativeUsage) that are commonly used to monitor the

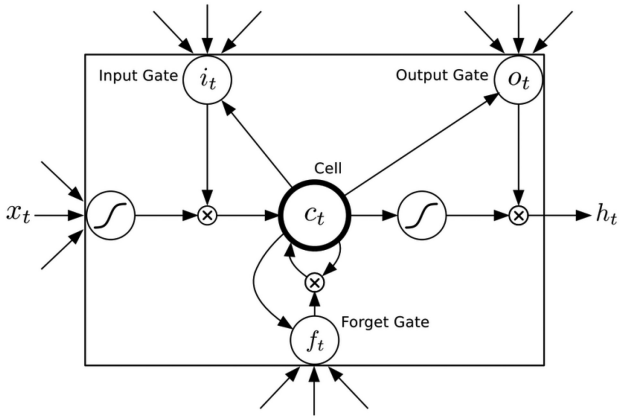


Fig. 2. Long Short-term Memory Cell.

performance of application. For JAVA applications, there are four regions (i.e., eden, survivor, old generation, and code cache regions) in the heap and two regions (i.e., meta and compressed class regions) in the native memory space. The detailed descriptions of the regions can be found in the OpenJDK documentation [46]. Each region is responsible for storing a unique type of data. Hence, we monitor the usage of each region. All twelve metrics shown in Table 1 can be obtained using performance monitoring applications without any instrumentation.

2.2 Architecture of Our Approach

To precisely model the behavior of an application, we need to capture the time dependency of the collected metrics (i.e., the dependency between current resource usages and past resource usages). LSTM neural network is well known to accurately capture the long term time dependency and model complex correlations in time-series data (e.g., the collected values of the monitoring metrics over time) [7, 43]. The LSTM neural network is a special kind of RNN that avoids the vanishing gradient problem existed in RNN [23]. The vanishing gradient problem means the loss of long-term dependencies introduced by the decaying gradient values. The LSTM neural network overcomes this problem by applying multiplicative gates that enforce constant error flow through the internal states of the LSTM memory cells. The key of the LSTM neural network is the internal states of its the LSTM memory cells. Figure 2 shows the structure of a LSTM memory cell. There are three multiplicative gates, i.e., input gate, forget gate, and output gate in the LSTM memory cells. The input gate is used to learn what new input information should be stored in the internal states of the memory cells. The forget gate controls how long the new input information should be stored and the output gate learns what parts of the stored information the memory cells should output. These three gates prevent the internal states of memory cells from being perturbed by irrelevant inputs and outputs, which enables the LSTM neural networks to capture long term time dependency. Thus, we choose to build a predictive the LSTM neural network to model the normal behaviors of an application.

The architecture of our approach is shown in Figure 3. Our approach contains four layers: one input layer, two hidden LSTM layers, and one output layer with three output branches. We build a deep LSTM network by stacking two LSTM layers, i.e., each neural unit in the lower LSTM hidden layer is fully connected to each unit in the higher LSTM hidden layer through feedforward connections. Prior studies show that stacking recurrent hidden layers enables a neural network to capture the structure of time-series data accurately [7, 22, 32, 41, 42, 43, 59]. Inspired by prior

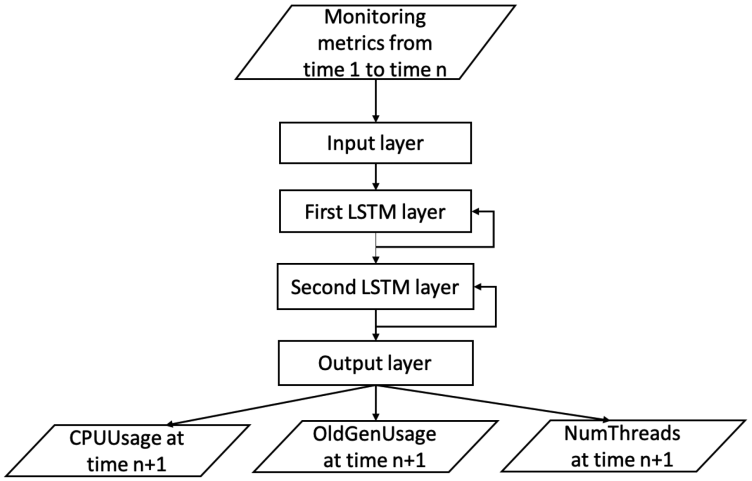


Fig. 3. The architecture of our approach.

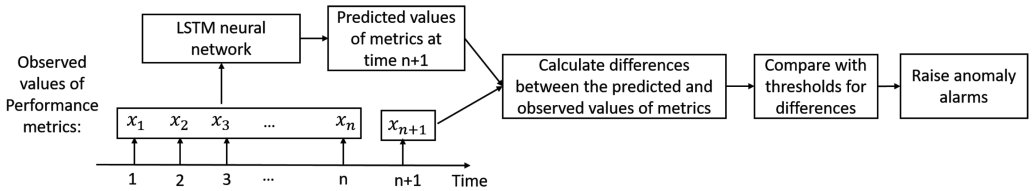


Fig. 4. An overview of our approach using the LSTM neural network to predict performance anomalies.

studies, we choose to use stacking LSTM layers to process the values of the monitoring metrics and model the normal behaviors of applications.

2.3 Training our Predictive LSTM Neural Network

As shown in Figure 4, the input to our LSTM neural network is a time series $x_1, x_2, x_3 \ldots, x_n$, where each entry $x_t, 1 \leq t \leq n$, is a 12-dimensional vector that contains the values of the monitoring metrics collected at time t. For each metric, we assign one neural unit in the input layer to take the value of the metric. To capture the behavior of an application, the next values (i.e., $x_{(n+1)}$) of the monitoring metrics should be predicted [42]. We train our LSTM neural network to predict the CPU usage, memory usage, and the number of threads, because these three metrics are common measures for the resource utilization of applications. For the memory usage, we choose to predict the *OldGenUsage* metric that measures the size of the old generation region in the heap. More specifically, the old generation region is used to only store stable objects [35, 46]. Thus, compared with other memory usage metrics, *OldGenUsage* contains the long-term memory usage of objects and is less fluctuating. During the training phase, our LSTM neural network tries to minimize the difference between the predicted values of the monitoring metrics and the real observed values of the monitoring metrics. We use only the values of the monitoring metrics collected from normal behaviors of applications to train our LSTM neural network. Thus, our LSTM neural network can capture normal behaviors of applications.

The output of our LSTM neural network is the predictions of *CPUUsage*, *OldGenUsage*, and *NumThreads*. The stacking LSTM layers, shown in Figure 3, extract the time dependency and correlations information from the input time-series values of the monitoring metrics. Then, the

extracted information is passed down to the three output branches and each branch predicts the value of a monitoring metric. Different from predicting all these three metrics together in a three-dimensional vector, predicting them separately provides information loss for each metric during the training process.

2.4 Performance Anomaly Prediction Using the Trained LSTM Neural Network

Performance bugs (e.g., memory leak and infinite loop bugs) do not always result in performance anomalies instantly. For example, it takes seven days for memory leak bug #8249 in Elasticsearch to occupy memory space and crash Elasticsearch as introduced in the bug report.[1] There is a time window, i.e., *pre-failure* time window, from the start of a performance bug until the performance anomaly occurs [11]. Therefore, we use the trained LSTM neural network to predict the future occurrences of performance anomalies by detecting whether the values of the monitoring metrics deviate from their normal behaviors.

Figure 4 shows an overview of our performance anomaly prediction approach. First, the trained LSTM neural network reads the collected time-series values of the monitoring metrics before time $n+1$, i.e., $x_1, x_2, x_3 \ldots, x_n$. Then, the LSTM neural network predicts the values of the monitoring metrics at time $n+1$, i.e., $\hat{CPUUsage}_{n+1}$, $\hat{OldGenUsage}_{n+1}$, and $\hat{NumThreads}_{n+1}$. Next, we calculate the differences between the three pairs of the predictions and the real observed values, i.e., ($\hat{CPUUsage}_{n+1}$, $CPUUsage_{n+1}$), ($\hat{OldGenUsage}_{n+1}$, $OldGenUsage_{n+1}$), and ($\hat{NumThreads}_{n+1}$, $NumThreads_{n+1}$). Our LSTM neural network is trained to capture the normal behaviors of an application. Thus, the prediction values are predicted following the knowledge learned from the normal behaviors. A large difference between any pairs of the predictions and the real observed values indicates that the monitored application deviates from its normal behaviors. We set up a threshold for the difference between each pair of the predictions and the real observed values. As shown in Figure 4, if the difference is greater than the threshold, then we think that the application is in a pre-failure state, and a performance anomaly warning is raised. Otherwise, the application is in a normal state, and no performance anomaly warning is raised. The design of the thresholds is explained in Section 4.1.

3 DATA COLLECTION

We evaluate our performance anomaly prediction approach using the data collected from two software systems: (1) Hadoop MapReduce sample applications that are JAVA open-source applications provided by Hadoop distribution [3]; and (2) Elatsticsearch, which is a JAVA open-source, distributed, and RESTful search engine [15]. To test the generalizability of our approach, we conduct experiments using seven different Hadoop MapReduce applications mentioned in Section 3.2. The studied seven Hadoop MapReduce applications have less than 1,000 lines of code. To evaluate our approach on a large-scale software system, we test our approach on Elasticsearch, which has more than 1.7 million lines of code. Elasticsearch is developed by over 1,300 developers and has more than 50,000 commits on GitHub. Furthermore, we test our approach on different released versions of Elasticsearch mentioned in Section 4.3.

Figure 5 shows an overview of our data collection approach. To train and evaluate our approach, we need to collect the metrics that track both normal and anomalous application behaviors. As shown in Figure 5, we first download the released versions of the studied applications. For collecting the values of the monitoring metrics for normal behaviors, we download the Elasticsearch version 5.3.0 and Hadoop applications that run on Hadoop MapReduce 3.1.3. We select these versions, because these versions are well tested and widely used in practice. Next, we collect the

[1] https://github.com/elastic/elasticsearch/issues/8249.

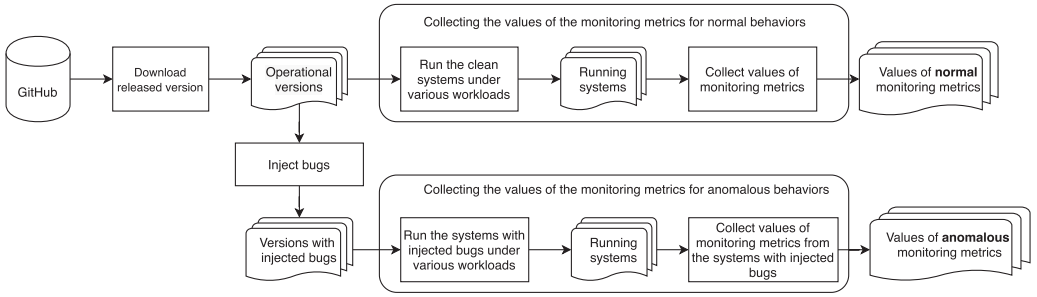


Fig. 5. An overview of our data collection approach.

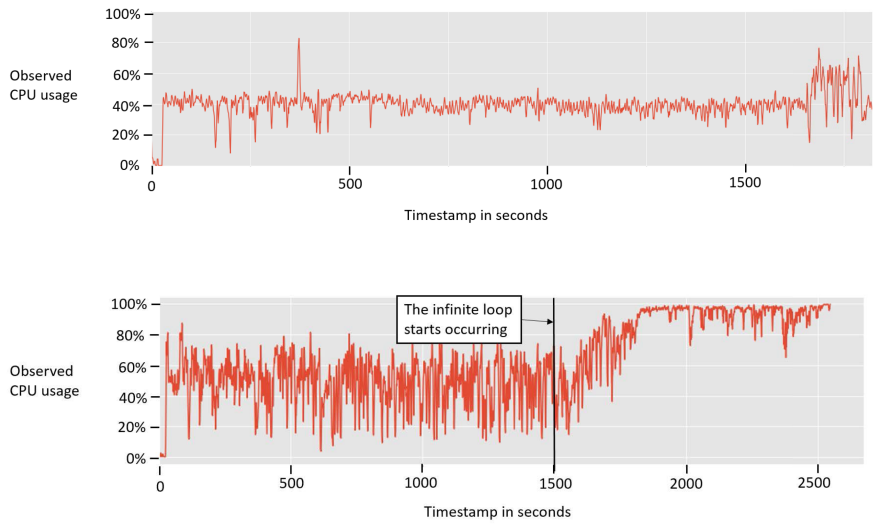


Fig. 6. An example of the values of the monitoring metrics for the normal and anomalous cases of the monitored Elasticsearch and Hadoop applications.

values of the monitoring metrics for normal behaviors by simulating the various workloads to the studied applications. Figure 6(a) shows an example of the values of the CPU usage for normal behaviors that are collected from Hadoop applications. Then, we inject bugs into the source code of the studied applications and obtain the buggy applications. Finally, we collect the values of the monitoring metrics for anomalous cases by simulating the workloads to the buggy applications. Figure 6(b) shows an example of the values of the CPU usage for anomalous cases that are collected from a buggy Elasticsearch with an injected infinite loop bug. As shown in Figure 6(b), after the occurrence of the infinite loop bug at the time mark of 1,500 s, the CPU usage of Elasticsearch starts increasing and reaches to 100%. We explain the bug injection and workload simulation steps as follows.

Bug injection: To simulate performance anomalies, previous studies inject bugs that are related to the CPU and memory usage into the source code [11, 19, 56, 66]. Inspired by the previous studies, we inject three types of performance bugs to the studied applications as follows.

- **Infinite loop**. We add infinite loop bugs that increase CPU consumption continuously by creating threads with infinite loops.

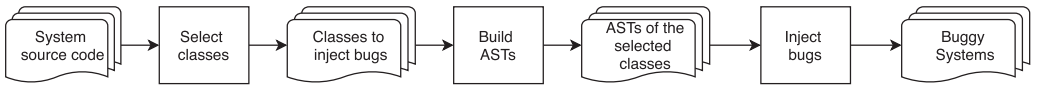


Fig. 7. An overview of our automatic bug injection approach.

```
41  public class RestAnalyzeAction {
            ...
78          public RestChannelConsumer
    prepareRequest(...) {
            ...
114         }
        ...
219 }
```

(a) The Java class before bug injection.

```
41  public class RestAnalyzeAction {
42          public static final ArrayList<Double>
    list = new ArrayList<Double>();
            ...
79          public RestChannelConsumer
    prepareRequest(...){
80          for (int i = 0; i< 300; i++){
            list.add(java.util.Random.nextDouble());
                }
            ...
114         }
    ...
219 }
```

(b) The Java class after bug injection.

Fig. 8. The source code a Java class before and after injecting a memory leak bug.

- **MemLeak**. We inject memory leak bugs, which continuously create objects and store them into static lists. Because the elements in static lists are not removed by the garbage collection process, the memory usage of the monitored JVM can keep on increasing.
- **Deadlock**. We insert buggy code to create threads with deadlock issues. The created threats deadlock each other.

We aim to inject the three types of performance bugs into different source code locations of the studied applications and obtain different buggy applications. Manually injecting performance bugs is error-prone and time-consuming. Therefore, we propose an automatic bug injection approach that injects bugs into the studied applications, as shown in Figure 7. We first select the JAVA classes in the source code to inject bugs. We only select JAVA classes that can be reached during run-time to inject performance bugs. Because the architectures of Hadoop MapReduce applications and Elasticsearch are different, we apply different strategies to select JAVA classes in the studies applications, as shown in Sections 3.1 and 3.2. We build the Abstract Syntax Trees (ASTs) of the selected classes, which represent the syntax structures of the source code of the classes. For each selected class, we analyze the AST and identify the location to inject performance bugs. Next, we inject the three types of performance bugs in the source code of the selected class. Figure 8(a) shows the source code of a Java class in Elasticsearch. Our approach analyzes the syntax structure of the class and identifies that the definitions of the *RestAnalyzeAction* class and the *prepareRequest* function starts at lines 41 and 78, respectively. Figure 8(b) shows the source code of the Java class after injecting a memory leak bug. A static list is injected in the *RestAnalyzeAction* class (i.e., line 42) and a for loop is injected at the beginning of the *prepareRequest* function (i.e., line 80) to add elements into the static list.

Running the normal/buggy applications under various workloads: We conduct all the workload simulation on a single machine with an 8-core Intel i7-4790 3.60 GHz CPU, 32 GB memory, and Ubuntu version 16.04.1. We deploy Elasticsearch and Hadoop MapReduce framework in a single-node cluster. We realize that our testbed environment is much simpler than the industrial environment in which a cluster could host thousands of nodes. As our approach works on the application-level resource usage metrics, a single-node cluster in our experiment design

Table 2. The Sample Applications Provided by Hadoop Distribution

Application	Description
bbp	An application that uses Bailey-Borwein-Plouffe algorithm to compute the exact digits of Pi.
pi	An application that estimates Pi using a quasi-Monte Carlo method.
multiFileWc	An application that counts the words in several input files separately.
wordCount	An application that counts the words in the input files.
wordMean	An application that counts the average length of the words in the input files.
wordMedian	An application that counts the median length of the words in the input files.
wordStandardDeviation	An application that counts the standard deviation of the length of the words in the input files.

can provide the same infrastructure as a multiple-node cluster for collecting the application-level metrics. We discuss how to use our approach in a large-scale cluster in the Discussion Section.

To demonstrate that our approach can be used in practice, we drive the subjected applications using various workloads that are observed in real-world scenarios as follows.

- For Elaticsearch, we use the datasets that are designed to conduct the macrobenchmarking test as the input workload. The macrobenchmarking test is a testing methodology used to evaluate the overall performance of applications [38].
- As shown in Table 2, the studied Hadoop MapReduce applications produce statistics about a set of text files. For example, the application wordMean counts the average length of the words in the input files. RandomTextWriter [2] is a text generation application that is provided by Hadoop to generate random large text files (e.g., files with 1 GB size). Hence, we generate the input datasets for the studied Hadoop MapReduce applications using the RandomTextWriter application.

Our approach collects the values of the monitoring metrics of the studied applications using a toolkit, called IBM Javametrics [25] that monitors resource usage of a JVM. IBM Javametrics can be configured to automatically store the values of the monitoring metrics, while other performance monitoring tools (e.g., tasks manager [65], or jconsole [47]) require manual effort to output the values of the monitoring metrics. We configure the JVM configurations of the studies applications to load the IBM Javametrics when the applications start running. Then, the loaded IBM Javametrics collect the resource usage of the applications at run-time with a configured sampling interval. We set the sampling time interval to 1 s in our experiments as used in the existing studies [11]. In the next sections, we describe the details for data collection in the studied applications.

3.1 Data Collection for Elasticsearch

We follow the bug injection approach shown in Figure 7 to obtain buggy versions of Elasticsearch. We first identify all the classes that handle HTTP requests. In Elasticsearch, the classes that handle HTTP requests act as entry points for processing HTTP requests. Hence, we can inject the performance bugs into the classes that handle HTTP requests and trigger the injected bugs once we send HTTP requests. For example, we obtain a buggy Elasticsearch by injecting an infinite loop bug into the class, *RestClusterStateAction.java*, which handles the HTTP requests for checking the

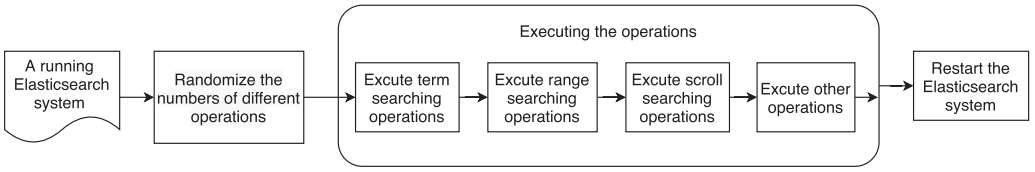


Fig. 9. A workload simulation example of applying a test suite to Elasticsearch.

cluster states. The injected bug creates a CPU consumption thread every time when the application handles a request for checking cluster state.

The API method, *controller.registerHandler()*, is used to register the classes for handling HTTP requests. Therefore, we find all the classes that handle HTTP requests by searching the keyword, *controller.registerHandler()*. We then manually check the classes identified by searching the keyword, *controller.registerHandler()*, and filter the classes that do not process HTTP requests. After the filtering, 90 classes that handle HTTP requests are identified. For each class, we obtain three buggy versions of Elasticsearch by injecting the three different types of performance bugs. In total, we obtain 90*3 (i.e., 270) buggy versions of Elasticsearch. After adding the one normal version of Elasticsearch, we have 90*3+1 (i.e., 271) versions of Elasticsearch in total.

We use Rally [14] to simulate real-world workloads. Rally is a macrobenchmarking framework for Elasticsearch, and it provides various macrobenchmarking test suites based on different datasets, such as questions and answers datasets from StackOverflow and the HTTP server log data dataset. In our experiment, a workload simulation is the execution of Elasticsearch under a macrobenchmarking test suite. Figure 9 shows an example of applying a test suite to Elasticsearch. As shown in Figure 9, the macrobenchmarking test suites simulate real-world scenarios of using Elasticsearch, such as executing term searching operations that search for different keywords or executing range searching operations that search for all documents created in a specific time window. Each test suite takes the numbers of different types of operations as configurable parameters. Hence, to be sure that the training data and testing data for our approach are collected under different workloads, we randomize the parameters of a test suite before applying it. Then, the different types of operations are executed sequentially. After executing the test suite, we restart the Elasticsearch and start applying the next test suite. We have 271 versions (i.e., 270 buggy versions and one normal version) of Elasticsearch. We find three stable test suites that can be applied to simulate workload. Thus, for each version of Elasticsearch (e.g., normal or buggy versions), we apply these three test suites in Rally. In total, we run 271*3 (i.e., 813) test suites. The duration of each test suite varies from 45 min to 2 h. It takes our machine around two months to finish running all the test suites and collecting the values of the monitoring metrics.

To generate data for anomalous situations, we keep on sending the HTTP requests that are handled by the injected classes to trigger the injected bugs. The starting time for sending the HTTP requests and the time intervals between every two requests are randomly selected and recorded. While simulating the workloads, we track the response times of Elasticsearch for processing the submitted HTTP requests. A performance anomaly is marked if the average response time is greater than a threshold (e.g., 100 ms) [11]. The marked performance anomalies are then used to evaluate the performance of our anomaly prediction approach.

3.2 Data Collection for Hadoop

We conduct experiments using seven sample applications that are provided by the Hadoop distribution as listed in Table 2. For each sample application, we use the approach shown in Figure 7 to automatically inject the three types of performance bugs into the Map class and obtain three

Table 3. The Parameters That Are Used to Build and Train Our LSTM Neural Network

Parameter	Value
Batch size	8
Number of input features	12
Number of steps	4
Number of hidden cells in LSTM layers	8
Number of epoches	20
Optimization function	Adam [34]
Loss function	Mean square error [67]

buggy applications. We have seven Hadoop applications, meaning that we obtain 7*3 (i.e., 21) buggy Hadoop applications and seven normal Hadoop applications in total. We then collect the values of the monitoring metrics following the data generation approach shown in Figure 5. For Hadoop applications (i.e., multiFileWc, wordCount, wordMean, wordMedian, and wordStandardDeviation), each simulated workload is 10GB text files. For Hadoop applications bbp and pi, each simulated workload is the number of the digits of π that needs to be calculated. To ensure that the training data and testing data for our approach are collected under different workloads, we generate different text files and the number of the digits of π whenever applying them to Hadoop applications.

We run the normal and buggy versions of these seven applications to collect the values of the monitoring metrics under normal and anomalous situations. We program the injected performance bugs to record their start time at run-time. We monitor the progress of each application and mark a performance anomaly when the application does not make any progress for processing the input data.

4 EVALUATION

In this section, we present the approach and the results of the studied research questions.

4.1 RQ1. What is the Performance of our Approach for Predicting Performance Anomalies?

Motivation: To test the ability of our approach for predicting performance anomalies, in this RQ, we measure the performance of our approach and compare it with the existing state-to-the-art approaches.

Approach: We collect the values of the monitoring metrics under normal and anomalous situations from the studied applications, as described in Section 3. We build our LSTM neural network using Keras [33]. Table 3 lists the parameters that are used to build and train our LSTM neural network. The detailed descriptions of the parameters can be found in the Keras documentation [33]. We train our LSTM neural network using the values of the monitoring metrics under normal behaviors, as introduced in Section 2.3. In the run-time monitoring phase, our LSTM neural network predicts the next values of the monitoring metrics by analyzing the current and past values of the monitoring metrics. In particular, our approach predicts the values for three monitoring metrics (i.e., $CPUUsage$, $OldGenUsage$, and $NumThreads$). Finally, we compare the predicted values with the actual observed values and raise anomaly warnings if the differences are larger than the predefined thresholds.

Measuring the performance of our approach. We use precision and recall to measure the performance of our approach for predicting performance anomalies. As shown in Figure 10, we

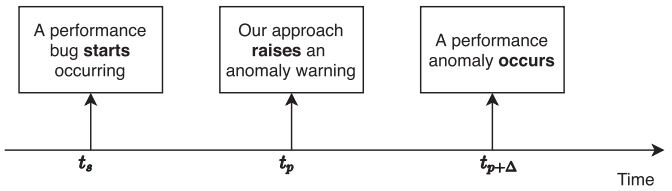


Fig. 10. An example of our approach making a true-positive prediction.

Table 4. Data Collected from the Workload Simulation of Elasticsearch

Datasets	Description	The number of the collected samples
Training	The the values of the monitoring metrics that are collected by running the normal version of Elasticsearch.	45,801
Validation	30% of the values of the monitoring metrics that are collected by running the buggy versions of Elasticsearch.	199,973
Testing	70% of the values of the monitoring metrics that are collected by running the buggy versions of Elasticsearch.	466,603

consider that our approach achieves a *true-positive* prediction if our approach raises an anomaly warning after the starting of the performance bug and before the occurrence of the performance anomaly. If our approach raises a warning, but there is no performance bug happening, then we consider that our approach makes a *false-positive* prediction. If our approach does not raise any warning, but the performance anomaly actually occurs, then we consider that our approach *fails* to predict the performance anomaly. Equations (1) and (2) show the computation for precision and recall. Precision measures the percentage of *true-positive* predictions among all the predictions made by our approach. Recall represents the percentage of the performance anomalies that can be predicted by our approach among all the anomalies in our experiment datasets:

$$Precision = \frac{Num_{tp}}{Num_{tp} + Num_{fp}}, \tag{1}$$

$$Recall = \frac{Num_{tp}}{Num_{tp} + Num_{fn}}, \tag{2}$$

where Num_{tp} and Num_{fp} are the number of true-positive (TP) and false-positive (FP) predictions that our approach makes. In addition, Num_{fn} is the number of false negatives (FN), i.e., the number of performance anomalies that our approach fails to predict.

Identifying the optimal thresholds for raising performance anomaly predictions. We use the values of the monitoring metrics under normal situations as our training datasets. We split the data for the values of the monitoring metrics under anomalous situations into the validation and testing datasets as shown in Tables 4 and 5. There are more buggy versions of the studied applications than the normal versions. Therefore, there are more data in the validation and testing datasets than the data in the training datasets. As mentioned in Section 3, we simulate different workloads to collect data. Thus, the training and testing data shown in Tables 4 and 5 are collected under different workloads, which enables us to measure the performance of our approach under different workloads from the training phase.

We record the time taken to train the LSTM neural networks on our experiment machine with an 8-core Intel i7-4790 3.60 GHz CPU and 32 GB memory. It takes 276 s (i.e., 4.6 min) and 240 s (i.e., 4 min) to train the LSTM neural networks for Elasticsearch and Hadoop applications,

Table 5. Data Collected from the Workload Simulation of Hadoop Applications

Datasets	Description	The number of the collected samples
Training	The values of the monitoring metrics that are collected by running all normal verions of the Hadoop applications.	36,625
Validation	30% of the values of the monitoring metrics that are collected by running the buggy versions of the Hadoop applications.	19,989
Testing	70% of the values of the monitoring metrics that are collected by running the buggy versions of the Hadoop applications.	46,641

respectively. We set up three thresholds for the values of the three monitoring metrics predicted by our LSTM neural network (i.e., $CPUUsage$, $OldGenUsage$, and $NumThreads$). Similar to the existing studies [7], we choose the thresholds by conducting sensitivity analysis to find the values that can yield to the maximum F-score on the validation datasets. F-score is a combination metric of precision and recall. Equation (3) shows the computation for F-score:

$$F - score = 2 * \frac{Precision * Recall}{Precision + Recall}. \tag{3}$$

We choose the thresholds using the validation dataset. Then, the selected thresholds are used on the testing datasets to predict performance anomalies and evaluate the performance of our approach.

Comparing with the existing state-to-the-art approaches. We select the UBL approach proposed by Dean et al. [11] as a baseline. The UBL baseline is unsupervised and is capable to predict performance anomalies by analyzing the values of the monitoring metrics. UBL baseline uses the Self Organizing Map (SOM) to capture the behaviors of applications. We build the SOM using the minisom python library [31]. For the configuration of the SOM, we use the best configuration presented by Dean et al. [11] in their experiments. We randomly initialize the weight vectors of the SOM as used in the UBL baseline [11]. We train the SOMs on our training datasets listed in Tables 4 and 5. The UBL baseline needs a threshold to differentiate the normal and anomalous values of the monitoring metrics. We determine the threshold by conducting the sensitivity analysis on the F-score using our validation datasets. Finally, we test the performance of the UBL baseline on our testing datasets.

In addition to the UBL baseline, we implement a simple baseline that raises performance anomaly warnings for an application by checking if the resource utilization of the application exceeds pre-defined thresholds. For example, VMware,[2] one of the top companies in providing cloud computing and virtualization software and services, suggests raising anomaly warnings once the CPU usage of a VM is above 95% [63]. We set up the thresholds for the values of the three monitoring metrics, i.e., $CPUUsage$, $OldGenUsage$, and $NumThreads$ and predict performance anomalies by checking if any of the values of these three metrics exceeds its threshold. We determine the thresholds for the values of these three monitoring metrics by maximizing the F-score on our validation datasets listed in Tables 4 and 5. We compare the performance of our approach with the performance of the two baselines (i.e., the UBL and the simple baselines).

Results: Our approach can predict various performance anomalies with 97–100% precision and 80–100% recall. Table 6 shows the performance of our approach and the baselines in predicting anomalies for Elasticsearch. As shown in Table 6, for **Elasticsearch**, our approach is able to achieve high precision of 98%, 97%, and 100% in predicting the performance anomalies that

[2] https://www.vmware.com/.

Table 6. The Performance of Our Approach and the Baselines in Predicting Performance Anomalies in the Studied Elasticsearch Experiments

	Precision			Recall			F-score		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	**98%**	**97%**	**100%**	98%	**100%**	87%	**98%**	**98%**	**93%**
The UBL baseline	97%	65%	25%	**100%**	93%	**94%**	98%	77%	39%
The simple baseline	76%	79%	78%	**100%**	**100%**	93%	86%	88%	85%

We highlight the results of the approaches that achieve the best performance.

Table 7. The Performance of Our Approach and the Baselines in Predicting Performance Anomalies in the Studied Hadoop Applications Experiments

	Precision			Recall			F-score		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	**100%**	**100%**	**100%**	**100%**	**100%**	80%	**100%**	**100%**	**89%**
The UBL baseline	41%	85%	70%	**100%**	**100%**	**100%**	58%	92%	82%
The simple baseline	67%	60%	71%	**100%**	**100%**	**100%**	80%	75%	83%

We highlight the results of the approaches that achieve the best performance.

are caused by the deadlock, infinite loop, and memory leak bugs, respectively. For **Elasticsearch**, our approach achieves the highest recall of 100% and 98% in predicting the performance anomalies that are caused by the infinite loop and deadlock bugs, respectively. Table 7 show the performance of our approach and the baselines in predicting anomalies for the studied Hadoop applications. For **Hadoop applications**, our approach achieves 100% precision and 100% recall in predicting the performance anomalies that are caused by the infinite loop and deadlock bugs.

The good performance regarding the infinite loop and deadlock bugs suggests that our approach models the behaviors of the normal CPU usage and the number of threads accurately. Conversely, We find that the lowest recall (i.e., 80% and 87%) of our approach occurs when the performance anomalies are caused by the memory leak bugs. The low performance for predicting the anomalies caused by the memory leak bugs can be explained by the high fluctuation of the memory usage. Memory usage depends on the size of the data that are processed by the applications. Because we run the studied applications to process large datasets (e.g., 10 GB text files for the Hadoop MapReduce applications), the memory usage of the applications is fluctuating. The high fluctuation of the memory usage impacts the ability of our approach to distinguish between the normal and anomalous behaviors of memory usage. Consequently, the high fluctuation reduces the recall of our approach in predicting the performance anomalies that are caused by the memory leak bugs.

Our approach consistently outperforms the UBL and the simple baselines in predicting performance anomalies in Elasticsearch and Hadoop applications. We highlight the results of the approaches that achieve the best performance in Tables 6 and 7. Our approach consistently achieves higher precision and F-score than the baselines. For **Elasticsearch**, our approach achieves an average precision of 98.3% in predicting anomalies that are caused by different types of performance bugs, while the UBL and the simple baselines achieve an average precision of

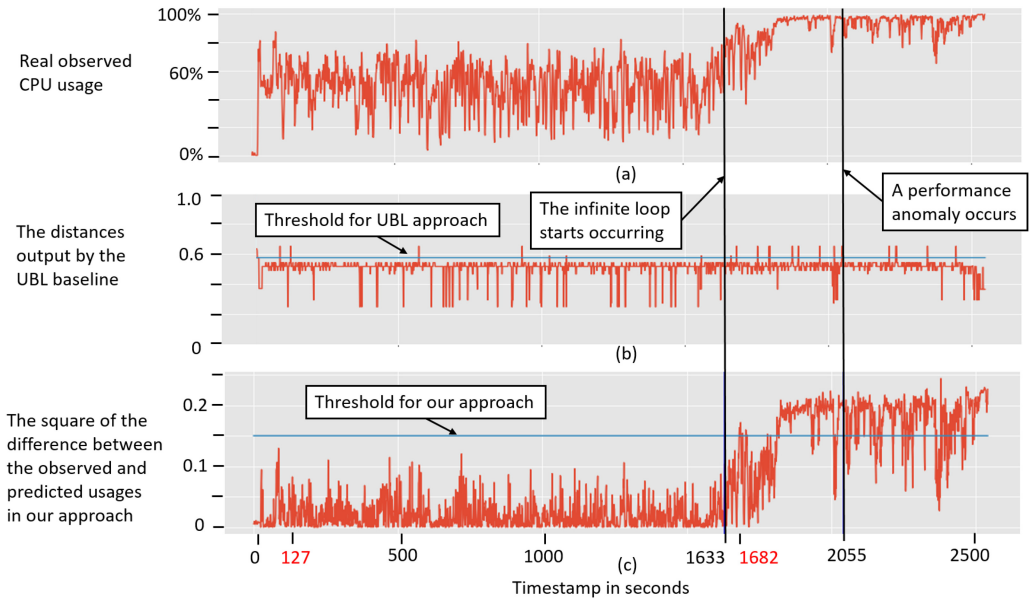


Fig. 11. An example of predicting anomalies that are caused by an infinite loop bug using the UBL baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red.

62.3% and 77.7%, respectively. For **Hadoop applications**, both our approach and the UBL baseline achieve high precision and recall in predicting the performance anomalies that are caused by the infinite loop bugs. The reason behind the high performance is that the CPU usage of the normal Hadoop applications does not experience high fluctuation during the execution of the map tasks. The map tasks execute the same operations on different chunks of data over time. The simple baseline achieves a low precision of 60% in predicting anomalies that are caused by the infinite loop bugs. One reason for the low performance is that the CPU usage threshold learned from the Hadoop validation dataset cannot scale to the CPU usage behaviors in the Hadoop testing dataset.

Anomaly prediction examples. The aforementioned results show that our approach outperforms the baselines, especially, our approach achieves higher precision in predicting performance anomalies. To obtain insights about the high precision of our approach, we provide examples to show the cases where the baselines raise false-positive predictions while our approach does not.

Figure 11 shows an example of using our approach and the UBL baseline to predict a performance anomaly that is caused by an infinite loop bug. As proposed by Dean et al. [11], for the values of the monitoring metrics collected at each second, the UBL baseline maps the values of the metrics into a winner neuron in the Self Organizing Map and calculates the distance between the winner neuron and its neighbor neurons. The distances for the collected values of the monitoring metrics are shown in Figure 11(b). If the distance is greater than a pre-defined threshold, then a performance anomaly warning is raised. As mentioned in Section 2, our approach raises performance anomaly warnings by checking the differences between the predicted values of the monitoring metrics and the real observed values of the metrics. The differences between the predicted and real observed values are shown in Figure 11(c). If the difference is greater than a threshold, then our approach raises a performance anomaly warning. As introduced in Section 4.1, the thresholds for our approach and the UBL baseline are selected by maximizing the F-score on the Elasticsearch validation dataset shown in Table 4.

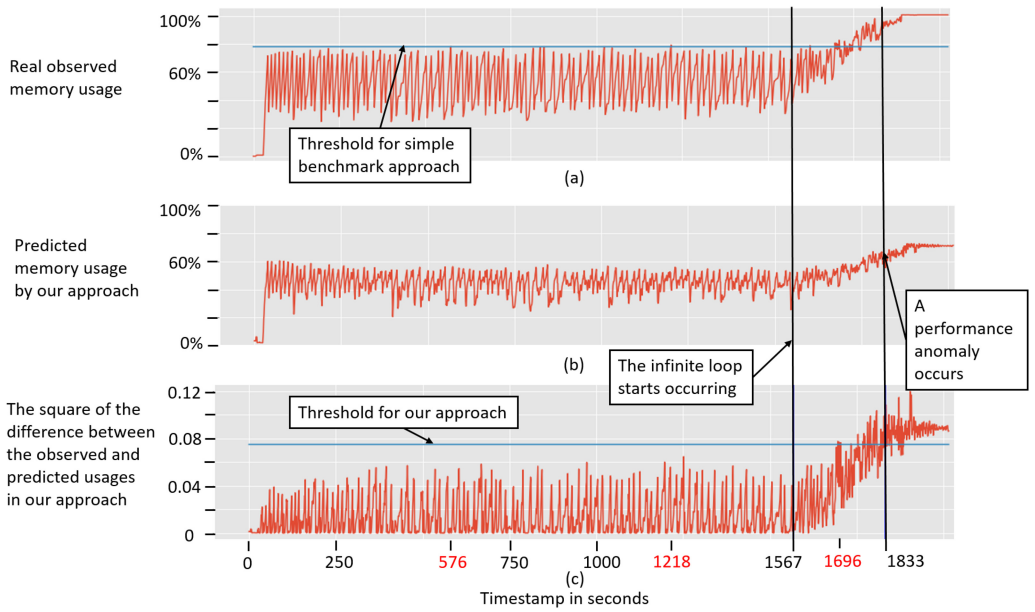


Fig. 12. An example of predicting anomalies that are caused by a memory leak bug using the simple baseline and our approach. Timestamps when our approach and the baseline approaches raise anomaly warnings are highlighted in red.

As shown in Figure 11, the CPU usage of Elasticsearch keeps on fluctuating from the beginning because of varying operations in the workload simulation. After an infinite loop bug starts to occur at the time mark of 1,633 s, the real observed CPU usage continues to increase and reaches to 100%. Both our approach and the UBL baseline make a true-positive anomaly prediction at the time mark of 1,682 s. However, the UBL baseline raises several false-positive predictions before the occurrence of the infinite loop bug because of the fluctuation of the CPU usage. For example, at the time mark of 127 s, the real CPU usage of Elasticsearch has a large fluctuation and the UBL baseline raises a false-positive prediction. Compared with the UBL baseline, no false-positive predictions are raised by our approach. Our LSTM neural network is not impacted by the fluctuations as the LSTM neural network can capture both the short fluctuations and the long-term changes.

Figure 12 shows an example of using our approach and the simple baseline to predict a performance anomaly that is caused by a memory leak bug. After a memory leak bug starts happening at the time mark of 1,567 s, the real observed memory usage of Elasticsearch continues to increase and reaches to 100% of the memory space. As shown in Figure 12, both our approach and the simple baseline are able to predict the performance anomaly around the time mark of 1,696 s. However, the simple baseline raises several false-positive predictions before the occurrence of the memory leak bug. The memory usage threshold learned from the validation dataset is 78%. At the time marks of 576 and 1,218 s, the memory usage of Elasticsearch increases and exceeds the threshold. Compared with the simple baseline, our approach is more robust to the fluctuations of the memory usage and achieves a higher precision.

The reasons behind the good performance of our approach. The aforementioned results and examples show that our approach achieves good performance in predicting performance anomalies. The explanation for the good performance of our approach is that the LSTM neural networks can incorporate the time dependency in time-series data (e.g., the values of the

monitoring metrics in our case). In addition, we use two stacking LSTM layers that have been observed to capture the time dependency accurately in the existing work [7, 41, 42, 43, 59].

From the aforementioned examples, we demonstrate that the LSTM neural networks can predict anomalies more accurately than the UBL baseline. We find that the LSTM neural networks are not impacted by the fluctuations in the values of the monitoring metrics as the neural networks can capture both the short fluctuations and the long-term changes. For example, the CPU usage of Elasticsearch has high fluctuations because of varying workloads. The high fluctuations result in the low precision of the UBL baseline in predicting anomalies (e.g., 25% precision in predicting anomalies caused by the memory leak bugs), while our approach achieves a precision of 100%.

Summary of RQ 1

Our approach can predict performance anomalies with 97-100% precision and 80-100% recall. Our approach outperforms the UBL and the simple baselines in predicting performance anomalies in the studied applications. By analyzing the anomaly prediction examples of our approach and the baselines, we observe that our approach can handle the fluctuations in the values of the monitoring metrics using the LSTM neural networks.

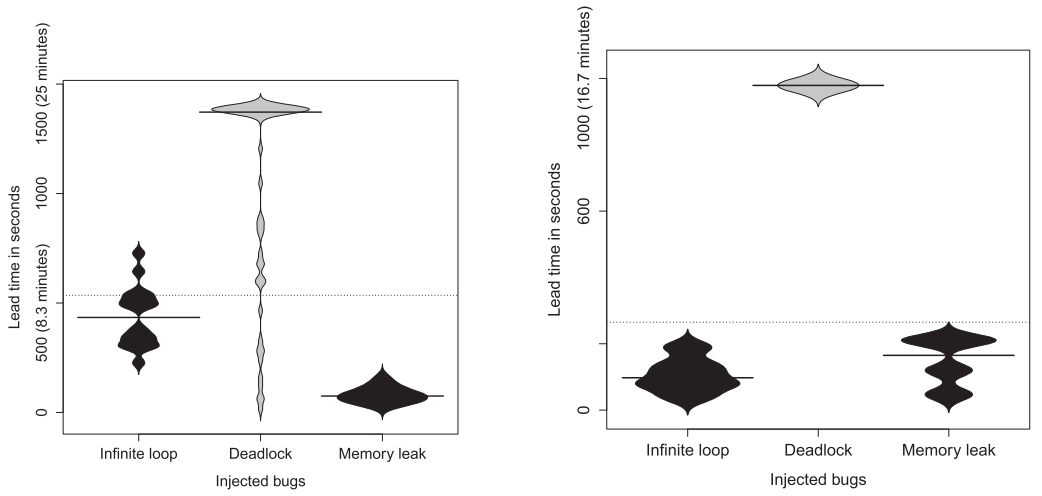
4.2 RQ2. How Early our Approach Can Raise a Performance Anomaly Warning before the Anomaly Occurs?

Motivation: To demonstrate that our approach is capable to predict performance anomalies in advance, in this RQ, we measure the lead time of our approach. Lead time calculates the amount of time that our approach can raise a performance anomaly warning before the anomaly occurs.

Approach. Dean et al. [11] mark the point when the performance anomalies occur by searching service level objective (SLO) violations (e.g., the average HTTP request response time is larger than a specific value). In particular, Dean et al. identify the performance anomalies when the average HTTP request response time is greater than 100 ms. We use the same threshold (i.e., 100 ms) to mark the performance anomaly occurrence in our experiment. For Elasticsearch, we mark a performance anomaly if the average response time for processing the HTTP requests is greater than 100 ms. For Hadoop applications, we mark a performance anomaly when the application does not make any progress for processing the input data. As shown in Figure 10, if a warning is raised at the second t_p and the real performance anomaly happens at the second $p_{p+\Delta}$, then the lead time is Δ s. As shown in Section 4.1, the precision of the baselines is low, meaning that the baselines tend to raise false anomaly warnings. The lead times achieved by the baselines are not valid, because many anomaly warnings are false warnings. Thus, we do not compare the lead times of our approach with the lead times of the baselines.

Results: Our approach predicts performance anomalies in advance with lead times that vary from 20 to 1,403 s (i.e., 23.4 min). Figures 13(a) and 13(b) show the lead times achieved by our approach for the studied applications. For **Elasticsearch**, the median lead time for predicting the anomalies that are caused by the deadlock bugs is 1,372 s (i.e., 22.8 min), with the maximum lead time of 1,403 s (i.e., 23.4 min). For predicting the anomalies introduced by the infinite loop bugs, our approach achieves a median lead time of 434 s (i.e., 7.2 min). Our approach achieves a median lead time of 75 s for predicting the anomalies that are caused by the memory leak bugs in Elasticsearch. We believe that the short lead times for the memory leak bugs are because of the high fluctuation of the memory usage at run-time. Thus, our approach can only predict an anomaly when the application is close to the anomalous state.

For **Hadoop applications**, our approach achieves a median lead time of 165 s (i.e., 2.8 min) for predicting the performance anomalies that are caused by the memory leak bugs. For the



(a) Lead times from Elasticsearch experiments

(b) Lead times from Hadoop experiments

Fig. 13. Beanplots of the lead times from Elasticsearch and Hadoop experiments. The dotted line represents the overall median lead time.

performance anomalies that are caused by the deadlock bugs, our approach is able to predict them with a median lead time of 979 s (i.e., 16.3 min).

Compared with the anomalies caused by the infinite loop and memory leak bugs, our approach can predict the anomalies caused by the deadlock bugs with longer lead times. The deadlock bugs continuously create threads with deadlock issues, and each thread has its own memory space. With more memory being consumed by the increasing number of threads, the studied applications become slower and cause performance anomalies to happen when system run out of memory. Our approach predicts the anomalies based on the anomalous behavior of the number of threads (i.e., the number of threads continues to increase). Our approach achieves long lead times (i.e., 979 s), because each thread only consumes a small amount of memory, and it takes a long time to create enough threads to affect the memory usage of the studied applications.

Our approach aims to predict performance anomalies and enable operators to prevent the predicted anomalies from happening. As mentioned in the existing study [57], local anomaly prevention actions, such as scaling VM resources, can be completed within one second and more costly anomaly prevention, such as live VM migration, takes 10–30 s to finish. The lead times achieved by our approach vary from 20 s to 1,403 s (i.e., 23.4 min), such lead times can be sufficient for the automatic prevention approaches [57] to take proper actions and prevent the anomalies from happening.

It should be noted that the achieved lead times can be impacted by the frequency of triggering the injected bugs during the workload simulation. For example, during the workload simulation of Elasticsearch, we send the HTTP requests to trigger the injected bugs. The more frequent we trigger the injected bugs, the quicker the performance anomalies can occur. Thus, the shorter lead times can be achieved by our approach. In addition, the selected thresholds can impact the lead times as follows. Using a small threshold to check the differences between the predicted values of the monitoring metrics and the real observed values of the metrics can improve the lead times, because a small deviation from the normal behaviors can trigger an anomaly warning. However, more false-positive predictions can be raised too when we apply a small threshold.

Summary of RQ 2

Our approach achieves lead times that vary from 20 to 1,403 seconds (i.e., 23.4 minutes) for predicting performance anomalies in our experiments. As suggested by the prior study [57], the achieved lead times are sufficient for automatic prevention approaches to take proper actions and prevent the predicted anomalies from happening.

4.3 RQ3. Could our Approach be used to Predict Anomalies Happened in the Real World?

Motivation: In the previous RQs, we measure the performance of our approach in predicting the performance anomalies caused by the injected bugs. However, the injected performance bugs might be more straightforward than the real-world performance bugs. Hence, in this RQ, we demonstrate the ability of our approach to predict the performance anomalies that are caused by real-world performance bugs and compare the performance of our approach with the baselines. In addition, we evaluate the run-time overhead of our anomaly prediction approach to test whether our approach is practical to predict anomalies in real-world systems.

Approach: To obtain performance bug reports, we search bug reports that contain keywords (e.g., memory leak, deadlock, and infinite loop) about performance bugs in the studied applications. No performance bug reports are returned after we search for the Hadoop MapReduce applications. The MapReduce applications are small (i.e., less than 1,000 lines of code) and are used as sample applications for learning purposes. Thus, it is reasonable that no performance bugs have been found in these sample applications. Therefore, we could not reproduce performance bugs in MapReduce applications.

For each identified performance bug report, we obtain the Elasticsearch version that contains the performance bug. Next, we identify the operations that can trigger the performance bug by exploring the bug reports and analyzing the source code of the buggy version. There are performance bugs that only occur under special workloads and running environments. For example, the infinite loop bug #30962[3] only happens in Windows platforms and the deadlock bug #36195[4] requires Elasticsearch to run on a multiple-node cluster. Due to our single-node experiment environment, we could not reproduce performance bugs that need intensive datasets or require the tested application to run on large clusters (e.g., deploying Elasticsearch to several virtual machines). Because of the limited and ambiguous information in bug reports, reproducing real-world performance bugs is a challenging and time-consuming task [50]. To this end, we successfully reproduce five performance bugs as listed in Table 8. We reproduce more memory leak bugs than the infinite loop and deadlock bugs, because memory leak bugs usually do not require a multiple-node cluster environment to reproduce.

To collect the values of the monitoring metrics in anomalous executions, we follow the same approach shown in Figure 5. We send HTTP requests that trigger the real-world performance bugs while running the macrobenchmarking test suites. Each Elasticsearch version has its supported test suites. In total, we apply 16 test suites to the five buggy Elasticsearch versions shown in Table 8. To collect more anomalous executions, we iterate the workload simulation five times [5]. To this end, we collect 16×5 (i.e., 80) anomalous Elasticsearch executions with performance anomalies. For the data collected from each workload simulation iteration (i.e., 16 anomalous Elasticsearch

[3] https://github.com/elastic/elasticsearch/issues/30962.
[4] https://github.com/elastic/elasticsearch/issues/36195.

Table 8. Summary of the Reproduced Real-world Performance Bugs

Bug ID	Description	Performance bug type	Elasticsearch version	Lines of code of Elasticsearch
#6553[5]	Out of memory issue occurs when using percolator queries.	Memory leak	1.2.1	381,159
#8249[6]	Heap usage grows when using cache keys.	Memory leak	1.3.4	414,081
#9377[7]	Elasticsearch hangs when using the collate option.	Deadlock	1.4.2	434,125
#24108[8]	High memory usage issue occurs when using nested queries.	Memory leak	5.5.0	2,988,321
#24735[9]	Thread falls into infinite loop when processing indices queries.	Infinite loop	5.3.1	2,549,236

Table 9. The Performance of our Approach and the Baselines in Predicting Performance Anomalies that are Caused by the Real-world Performance Bugs

	Precision			Recall			F-score		
	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak	Deadlock	Infinite loop	Memory leak
Our approach	**100% ±0%**	**100% ±0%**	**95% ±9%**	87% ±9%	**100% ±0%**	**100% ±0%**	**93% ±5%**	**100% ±0%**	**97% ±5%**
The UBL baseline	49% ±3%	59% ±4%	71% ±15%	**100% ±0%**	**100% ±0%**	**100% ±0%**	66% ±2%	74% ±3%	82% ±11%
The simple baseline	50% ±0%	83% ±24%	68% ±19%	**100% ±0%**	**100% ±0%**	**100% ±0%**	67% ±0%	89% ±16%	79% ±13%

We highlight the results of the approaches that achieve the best performance.

executions), we evaluate the precision, recall, and F-score of our approach and the baselines in predicting anomalies. We use the same prediction models and thresholds that are trained and validated on the datasets shown in Table 4. After five iterations, we calculate the average and the standard deviation of the precision, recall, and F-score of our approach and baselines in predicting performance anomalies as shown in Table 9. To evaluate the run-time overhead of our anomaly prediction approach, we measure the amount of time, CPU usage, and memory usage that our approach takes to make anomaly predictions. We conduct the experiments on our experiment environment with an 8-core Intel i7-4790 3.60 GHz CPU and 32 GB memory.

Results: Our approach achieves high precision and recall in predicting the performance anomalies that are caused by the five reproduced performance bugs in Elasticsearch. As shown in Table 9, our approach achieves an average of 100% precision and 87% recall in predicting performance anomalies that are caused by the deadlock bugs. In addition, our approach achieves average 100% and 95% precision in predicting performance anomalies caused by the infinite loop and memory leak bugs.

Our approach achieves higher average performance with less deviation compared with the baselines. As shown in Table 9, our approach consistently achieves higher F-score in

[5]https://github.com/elastic/elasticsearch/issues/6553.
[6]https://github.com/elastic/elasticsearch/issues/8249.
[7]https://github.com/elastic/elasticsearch/issues/9377.
[8]https://github.com/elastic/elasticsearch/issues/24108.
[9]https://github.com/elastic/elasticsearch/issues/24735.

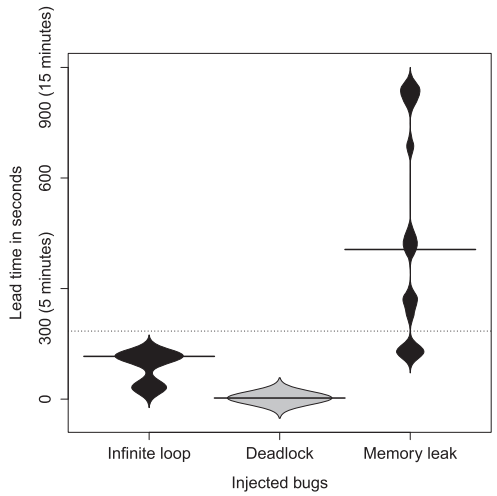


Fig. 14. Beanplots of the lead times for predicting the performance anomalies caused by the five reproduced performance bugs in Elasticsearch. The dotted line represents the overall median lead time.

predicting performance anomalies that are caused by the different real-world performance bugs than the baselines. In addition, our approach achieves less deviation. Less deviation means that our approach achieves more stable performance in predicting performance anomalies than the baselines. For example, the deviation of the F-score of our approach in predicting anomalies that are caused by memory leak bugs is 0.05, while the deviation of the UBL approach is 0.11.

Our approach predicts the performance anomalies caused by the real-world performance bugs with lead times that vary from 2 to 846 s (i.e., 14.1 min). As shown in Figure 14, the median lead time for predicting the anomalies that are caused by the infinite loop bug is 116 s (i.e., 1.9 min). For predicting the anomalies that are caused by the memory leak bugs, our approach achieves a median lead time of 406 s (i.e., 6.7 min), with the maximum lead time of 846 s (i.e., 14.1 min). Our approach achieves the median lead time of 3 s for predicting the anomalies that are caused by the deadlock bug.

We analyze the reason behind the short lead times for predicting the anomalies caused by the deadlock bug #9377. The deadlock bug #9377 causes the Elasticsearch to hang with the searching threads deadlocking each other. The searching threads in Elasticsearch are responsible for processing document searching requests. After all the searching threads deadlock each other, Elasticsearch stops processing document searching requests and puts searching requests into the waiting queue. After the waiting queue becomes full, Elasticsearch starts throwing out exceptions and rejecting more searching requests. Because Elasticsearch is used to search documents in practice, during the workload simulation, intensive document searching requests (e.g., hundreds in a second) are sent to Elasticsearch. In our experiments, we use the default queue size of 1,000 [16]. After the deadlock bug occurs, the waiting queue of Elasticsearch is filled with a thousand of searching requests within a few seconds, and Elasticsearch starts rejecting additional searching requests. The 100% precision shown in Table 9 means that our approach is able to predict the performance anomalies caused by the deadlock bug #9377 when the anomalies happen in a few seconds after the occurrence of the performance bug.

We manually analyze the *false-positive* predictions that are raised by our approach. Figure 15 shows a false-positive prediction example when we test our approach using the data collected from the memory leak bug #24108. We run Elasticsearch under a workload that requires Elasticsearch

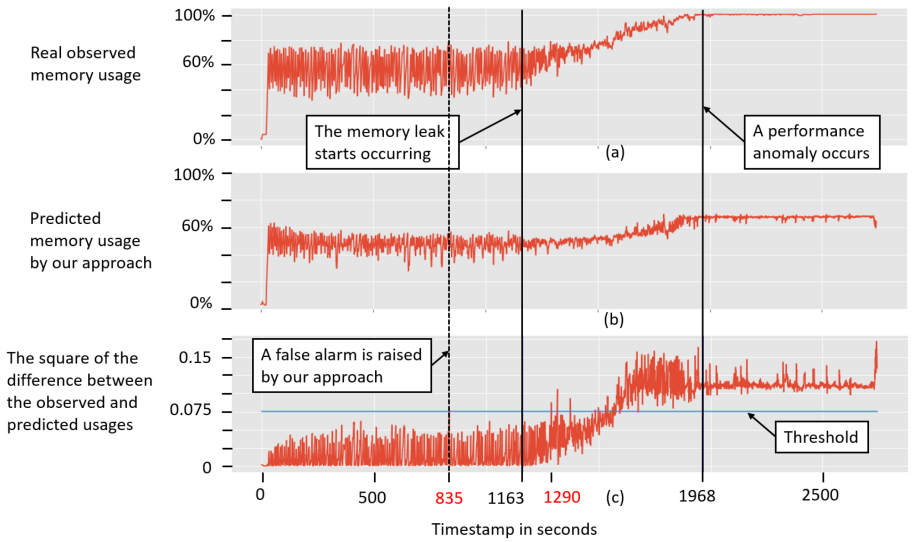


Fig. 15. A false-positive prediction that is raised by our approach. Timestamps when our approach raises anomaly warnings are highlighted in red.

Table 10. The Measurements of the Overhead of Our Approach

Number of samples	Time taken (in seconds)	Average CPU usage	Maximum CPU usage	Average memory usage	Maximum memory usage
141,647	36	21.8%	51.6%	2.0%	2.1%

to load a large number of documents into the memory. f Thus, the real observed memory usage fluctuates from the beginning, and it is difficult to predict the fluctuating value precisely. As shown in Figure 15(c), our trained LSTM neural network can predict fluctuating memory usage with small differences. After a memory leak bug starts to occur at the time mark of 1,163 s, the real observed memory usage continues to increase and reaches to 100% usage of the memory space. A performance anomaly is observed at the time mark of 1,968 s based on the increase in the average response times of the HTTP requests (as described in Section 3.1). As shown in Figure 15(b), our approach predicts that memory usage should stop growing at around 70%. The differences between the observed and predicted memory usages start increasing before the occurrence of the anomaly. At the time mark of 1,290 s, the difference exceeds the threshold (i.e., 0.075), and our approach raises a true anomaly warning. As introduced in Section 4.1, the threshold (i.e., 0.075) is selected by maximizing the F-score on the Elasticsearch validation dataset shown in Table 4. However, before the memory leak bug starts to occur, the difference between the observed and the predicted memory usages exceeds the threshold (i.e., 0.075) at the time mark of 835 s, and our approach raises an anomaly warning. There is no performance bug occurring at the time mark of 835 s and Elasticsearch is under normal executions. Hence, the anomaly warning is false, and our approach makes a false-positive prediction.

The run-time overhead of our performance anomaly prediction approach is feasible for real-world systems. As shown in Table 10, there are 141,647 samples in the monitoring data collected from the replicated performance bugs. For each sample (i.e., the values of the monitoring metrics collected at each second), our approach processes the collected values of the monitoring

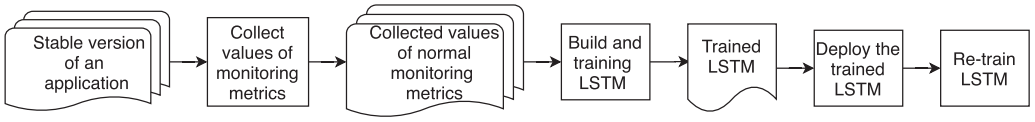


Fig. 16. The overview of implementing our approach to monitor an application.

metrics and predicts if a performance anomaly would happen. Our approach takes 36 s to process the 141,647 samples and predict performance anomalies that are caused by the real-world performance bugs. On average, our approach needs 36/141,617 s (i.e., 0.25 ms) to predict anomalies using the values of the monitoring metrics collected at every second. The average CPU usage of our approach for predicting anomalies is 21.8% on our machine. There are only 8 cores in our CPU, which means that our approach needs a average of $8 \times 21.8\%$ (i.e., 2) CPU cores to predict anomalies. The average memory usage of our approach is 32 GB×2.0% (i.e., 320 MB). More powerful CPUs and larger memory size than our experiment machine are used in the industrial environments. For example, in Amazon's Elastic Compute Cloud (EC2), each node (i.e, virtual machine) can have up to 16 cores of 2.3 GHz AWS Graviton CPU and 32 GB memory [1]. In the industrial environments, our approach can be assigned to separate CPU cores from the monitored application and do not impact the executions of the monitored application. Thus, our anomaly prediction approach is practical to predict anomalies in real-world systems.

Summary of RQ 3

Our approach achieves higher average performance with less deviation compared with the baselines in predicting the performance anomalies caused by the five real-world performance bugs. Our approach achieves lead times that vary from 2 to 846 seconds (i.e., 14.1 minutes). In addition, our approach takes an average of 0.25 milliseconds, 320MB memory, and 25% CPU usage to predict anomalies at every second.

5 DISCUSSION

In this section, we discuss the implementation of our approach to monitor applications in the real-world industrial environment, the usage scenarios of our approach, and the limitations of our approach.

5.1 Implementing Our Approach to Monitor Applications in the Industrial Environment

Data collection. Figure 16 shows the overview of implementing our approach to monitor an application. As shown in Figure 16, the first step to implement our anomaly prediction approach is to collect data to train the LSTM neural networks. For an application, the perfect training data should be collected from monitoring the latest stable version of the application. Human operators can manually examine the collected data to make sure that no performance anomaly happened during the data collection.

Building and training LSTM neural networks. After data collection, the parameters for building and training the LSTM neural networks should be adjusted based on the collected data as shown in Figure 16. For example, the number of input features should be the same as the number of monitoring metrics in the collected data. Once the parameters are decided, the LSTM neural networks can be trained offline using the collected data.

Deploying and re-training LSTM neural networks. The trained LSTM neural networks can be deployed on the platform (e.g., VMs in a cloud infrastructure), which hosts the application to be monitored. After the deployment, the LSTM neural networks can track the values of the

monitoring metrics of the application as input and predict performance anomalies. In addition, the behaviors of the application might change because of the application updating. To adapt the LSTM neural networks to the new behaviors, the LSTM neural networks should be re-trained using the data collected from the new stable versions of the application as shown in Figure 16. However, frequently re-training can decrease the performance of the LSTM neural networks if the neural networks are trained on a non-stable version, i.e., a version that has buggy behaviors but are not realized by developers.

5.2 The Usage Scenarios of Our Approach

We deploy our approach to monitor the application-level metrics. In this section, we describe two scenarios of deploying our approach.

Scenarios when an application runs on a multiple-node cluster. Applications can be hosted on a cluster with thousands of nodes (e.g., virtual machines). For monitoring an application running on a large-scale cluster, we can first collect application-level metrics of the instance of the application (i.e., the application process) running on each node (e.g., virtual machine) in the cluster. Then, we can train the LSTM neural network using the collected application-level metrics values. Next, our approach can be deployed into each node in the cluster. For every node, our approach monitors the instance of the application running on the node and predicts anomalies for the instance of the application. By predicting anomalies for each instance of the application, our approach can help developers and operators to locate the anomalous instance of the application in the cluster.

Scenarios when several applications are running in the same system. In an industrial environment, there might be several applications running on the same machine or VM. The co-existing applications share computation resources, such as CPU and memory. The anomalous behaviors of one application can affect the executions of other co-existing applications. For example, a memory leak bug in an application can cause the system running out of memory and affect the executions of other applications in the system. For the scenarios when several applications are running in the same system, operators can deploy multiple instances of our approach. More specifically, each instance monitors an application and predicts the performance anomalies for the application. By monitoring each application, our approach can predict which application is the anomalous application. For example, applications A and B are running in the same system and there is a memory leak happening in application B. After deployment of separate instances of our approach to each application, our approach can predict performance anomalies in application B based on its increasing memory usage.

5.3 The Limitations of Our Approach

Our approach captures the normal behaviors of applications using the LSTM neural networks. In our experiments, we observe that our approach achieves good performance in predicting performance anomalies by checking whether applications deviate from the captured normal behaviors. However, every approach has its limitations. In this section, we summarize and discuss the limitations of our approach.

Our approach may not be able to differentiate non performance anomalies that deviate from normal behaviors from performance anomalies. For example, a spontaneous increase in the number of documents loading requests to Elasticsearch can suddenly increase the CPU and memory usages of Elasticsearch . In our experiments, we observe that our approach can handle the fluctuations in the monitoring metrics (e.g., CPU and memory usages). However, if a workload change results in large fluctuations in the monitoring metrics, our approach might raise a false

performance anomaly warning. For example, our approach makes a false-positive prediction when the memory usage is fluctuating heavily as shown in the example in Section 4.3.

Our approach has limitations to automatically adapt to the new normal behaviors of applications. The LSTM neural networks capture only the normal behaviors that are included in the training dataset. However, the behaviors of applications could change over time because of the updates of the application. In our approach, the LSTM neural networks cannot automatically adapt to the new behaviors of applications at the run-time monitoring phase. The LSTM neural networks are required to be re-trained offline to learn the new behaviors of applications.

6 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study.

Threats to external validity is related to the generalizability of our results with respect to other project settings. To ensure the generalizability of our approach, we conduct experiments on two JAVA software systems. These two software systems have different architectures and belong to different domains. However, studying software systems that are programmed in other languages can be useful to augment the generalizability of our approach.

Threats to internal validity concern the uncontrolled factors that may affect the experiment results. One internal threat to our results is that the injected performance bugs are simpler than real-world performance bugs. To mitigate this threat, we test our approach on real-world performance bugs by reproducing the performance bugs of Elasticsearch. We manage to reproduce five real-world performance bugs of Elasticsearch. The obtained results show that our approach can accurately predict the performance anomalies that are caused by real-world performance bugs. However, further studies can explore reproducing more performance bugs, such as concurrent performance bugs. The time interval that is used to collect the values of the monitoring metrics of the studied systems can affect our results. Different time intervals might affect the ability of our approach for monitoring software systems and predicting performance anomalies. We apply one second time interval following the existing work [11]. However, further studies can explore to set the best time intervals for different software systems. In addition, the thresholds that are used to raise performance anomaly warnings can affect our results. There is a tradeoff between precision and recall. Using a smaller threshold can improve the recall of our approach, because small deviations from the normal behaviors can raise performance anomaly predictions. However, a smaller threshold can affect the precision of our approach, since the predictions raised from small deviations might be false-positive predictions. In our experiments, we determine the values of the thresholds by selecting the values that achieve the best F-score on the validation datasets.

7 RELATED WORK

In recent years, studying operational and performance anomalies has been of great interest to researchers. In this section, we discuss the related work concerning operational anomaly detection, performance anomaly detection, and performance anomaly prediction.

7.1 Operational Anomaly Detection

Operational anomalies are related to exceptional workflows. For example, a NameNode in Hadoop Distributed File System is not updated after deleting a data block [69]. Generally speaking, operational anomaly detection approaches analyze console logs of a software system to monitor the execution flow of the system in two steps: (1) Parsing unstructured logs to produce structured log events; and (2) Operational anomaly detection using structured log events.

Step 1: Parsing unstructured logs to produce structured log events. Logs record the runtime information (e.g., execution traces and the internal states of programs) during the execution

of a software system. However, the logs are usually unstructured and free-form text. Before feeding log messages into log analyzing approaches, the unstructured log messages need to be parsed to produce structured log events. For example, the log printing statement *printf("Start transaction %d.", id)* contains a constant string, i.e., *Start transaction*, and a variable transaction parameter, i.e., *id*. Log parsing is used to identify the log event *Start transaction **, where $*$ stands for the place holder for variables (i.e., parameter values). Various log parsing approaches have been proposed in recent years, such as References [12, 18, 21, 40, 58, 62, 68, 72].

Step 2: Detecting operational anomalies using structured log events. Various supervised and unsupervised machine learning approaches are proposed to detect operational anomalies. For example, Yen et al. [70] propose an unsupervised clustering approach that identifies malware infections and policy violations based on the application-specific features extracted from logs. Xu et al. [68] propose a PCA-based unsupervised approach to detect the anomalies from the structured log events. Fu et al. [18] use Finite State Automation, an unsupervised machine learning technique, to model the normal workflow from system logs and detect the anomalies based on the learned workflows. Lou et al. [39] group the structured log events into different groups and mine program invariants from each group to represent the system workflows. The mined invariants are then used to detect anomalies from logs. Lin et al. [37] propose a cluster-based approach that checks the occurrence of log sequences to detect anomalies. Du et al. [13] propose a LSTM neural network to model structured log events as natural language sequences and learn log patterns from the normal executions to detect anomalies. In addition to unsupervised algorithms, Liang et al. [36] apply a supervised learning algorithm, i.e., Support Vector Machine, to detect operational anomalies. Mike et al. [8] use a decision tree to detect and diagnose system failures by analyzing the logs from user requests. Different from the aforementioned *operational anomaly detection* approaches, our approach aims to predict performance anomalies for software systems.

7.2 Performance Anomaly Detection

Performance anomalies are related to exceptional resource utilization. Researchers propose various approaches to automatically detect performance anomalies by analyzing logs and runtime metrics (e.g., resource utilization and network traffic statistics) of the monitored systems. Munawar et al. [45] employ a linear regression model to identify correlations among measurement metrics, such as the correlation between memory usage and the number of live threads. The measured correlations are then used to characterize normal behaviors and identify anomalous states. In addition to applying linear regression, researchers explore various statistical techniques, such as multi-variable regression [29], locally-weighted regression [44], auto-regressive regression [27], and Gaussian mixture models [20] to capture correlations among measurement metrics. In contrast to the above approaches, our approach can predict future anomalies instead of detecting anomalies after the occurrence.

In addition to analyzing correlations among metrics, Shen et al. [51] propose a reference-driven performance anomaly detection approach by checking how metrics differ from the ideal behavior. Powers et al. [48] explore different statistical learning methods to detect performance violations in a software system. Cohen et al. [10] extract signatures from the labeled failure data to detect recurrent problems. Later, Bodik et al. [6] improve the signature extraction approach by applying feature selection techniques. Cherkasova et al. [9] use a regression-based transaction model with application performance signatures to distinguish between performance anomalies and system changes. Jiang et al. [28] propose an approach to detect system failures based on the entropy of the clustered system metrics. Roy et al. [49] introduce an approach to find performance anomalies by checking a subset of correlated system metrics. Stewart et al. [53] apply a transaction mix model to predict the performance of a system given a certain workload. Anomalies are detected if

the observed performance is different from the predicted performance. Wang et al. [64] propose an entropy-based approach to detect anomalies by analyzing the distributions of the metrics. Different from the aforementioned anomaly detection studies, our approach aims to predict performance anomalies in advance and can potentially enable operators or automatic prevention approaches to take proactive actions.

7.3 Performance Anomaly Prediction

Prediction approaches intend to predict anomalies before they occur, while detection approaches identify anomalies when the anomalies happen. Existing studies propose approaches that predict performance anomalies in virtual machines (i.e., predicting VM-level anomalies). For example, Williams et al. [66] propose a supervised approach to predict failures in a distributed system by analyzing VM-level resource usage metrics collected from every node in the distributed system. Gu et al. [19] combine Markov models and Bayesian classification models to process the VM-level resource usage metrics and predict performance anomalies. Huang et al. [24] apply recurrent neural networks to predict performance anomalies in distributed systems. In contrast to our approach, the aforementioned approaches [19, 24, 66] require labeled training datasets that represent normal and anomalous behaviors of the studied systems. Dean et al. [11] propose an unsupervised behavior learning approach to predict the failures of virtual machines by analyzing the monitoring data collected from VMs in cloud infrastructures. Tan et al. [56] propose a context-aware anomaly prediction approach to predict the failures of virtual machines in large-scale hosting infrastructures.

Different from our work, the aforementioned approaches [11, 56] provide VM-level performance anomalies prediction. Our approach can issue application-level performance anomaly warnings that predict the anomalous applications in a virtual machine in advance. In our experiments, we observe that our approach is able to handle the fluctuations in the monitoring data using the LSTM neural networks. We find that our approach achieves more accurate results in predicting application-level performance anomalies than the approach proposed by Dean et al. [11]. By providing more accurate predictions, our approach can be more practical in the industrial production environments.

8 CONCLUSION

In this article, we present an approach that predicts performance anomalies in software systems. First, our approach uses LSTM neural networks to capture the normal behaviors of software systems. Then, our approach predicts performance anomalies at run-time by checking the early deviations from the normal behaviors that are expected from the LSTM neural networks. We conduct extensive experiments using two real-world software systems (i.e., Elasticsearch and Hadoop applications) to evaluate our approach. In addition, we demonstrate the ability of our approach to predict the performance anomalies that are caused by real-world performance bugs. The obtained results show:

- Our approach outperforms the baselines and predicts various performance anomalies with 97–100% precision and 80–100% recall.
- Our approach predicts performance anomalies in advance with lead times that vary from 20 to 1,403 s (i.e., 23.4 min).
- Our approach achieves 95–100% precision and 87–100% recall with lead times that vary from 2 to 846 s (i.e., 14.1 min) for predicting the anomalies that are caused by the five real-world performance bugs in Elasticsearch.

Overall, the results of the experiments show that our approach achieves high performance in predicting anomalies that are caused by both the injected bugs and real-word performance bugs. In the future, we plan to collect data from more real-world software systems, reproduce more real-world performance bugs (e.g., concurrent performance bugs), and compare the performance of our approach with more existing approaches.

REFERENCES

[1] Amazon. [n.d.]. Amazon EC2 Instance Types. Retrieved from https://aws.amazon.com/ec2/instance-types/.

[2] Apache. [n.d.]. Apache Hadoop RandomTextWriter application. Retrieved from https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/RandomTextWriter.html.

[3] Apache. [n.d.]. Apache Hadoop System. Retrieved from http://hadoop.apache.org/.

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using magpie for request extraction and workload modelling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Vol. 4. 18–18.

[5] Stefan Berner, Roland Weber, and Rudolf K. Keller. 2005. Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*. 571–579.

[6] Peter Bodík, Moises Goldszmidt, and Armando Fox. 2008. HiLighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *Proceedings of the Systems Modeling Language Conference (SysML)*.

[7] Sucheta Chauhan and Lovekesh Vig. 2015. Anomaly detection in ECG time signals via deep long short-term memory networks. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA'15)*. IEEE, 1–7.

[8] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing*. IEEE, 36–43.

[9] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2008. Anomaly? Application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*. IEEE, 452–461.

[10] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 105–118.

[11] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th International Conference on Autonomic Computing*. ACM, 191–200.

[12] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *Proceedings of the IEEE 16th International Conference on Data Mining (ICDM'16)*. IEEE, 859–864.

[13] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.

[14] Elastic. [n.d.]. Rally. Retrieved from https://github.com/elastic/rally.

[15] ElasticSearch. [n.d.]. Elasticsearch. Retrieved from https://www.elastic.co.

[16] ElasticSearch. [n.d.]. Elasticsearch Reference. Retrieved from https://www.elastic.co/guide/en/elasticsearch/reference/5.3/modules-threadpool.html.

[17] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.* 30, 4 (2012), 1–35.

[18] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*. IEEE, 149–158.

[19] Xiaohui Gu and Haixun Wang. 2009. Online anomaly prediction for robust cluster systems. In *Proceedings of the IEEE 25th International Conference on Data Engineering*. IEEE, 1000–1011.

[20] Zhen Guo, Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. 2006. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 259–268.

[21] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the IEEE International Conference on Web Services (ICWS'17)*. IEEE, 33–40.

[22] Michiel Hermans and Benjamin Schrauwen. 2013. Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*. MIT Press, 190–198.
[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.
[24] Shaohan Huang, Carol Fung, Kui Wang, Polo Pei, Zhongzhi Luan, and Depei Qian. 2016. Using recurrent neural networks toward black-box system anomaly prediction. In *Proceedings of the IEEE/ACM 24th International Symposium on Quality of Service (IWQoS'16)*. IEEE, 1–10.
[25] IBM. [n.d.]. IBM Javametrics. Retrieved from https://developer.ibm.com/javasdk/application-metrics-java/.
[26] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. 2006. Discovering likely invariants of distributed transaction systems for autonomic system management. In *Proceedings of the IEEE International Conference on Autonomic Computing*. IEEE, 199–208.
[27] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. 2006. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Trans. Depend. Secure Comput.* 3, 4 (2006), 312–326.
[28] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. 2009. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 285–294.
[29] Miao Jiang, Mohammad A. Munawar, Thomas Reidemeister, and Paul A. S. Ward. 2009. System monitoring with metric-correlation models: Problems and solutions. In *Proceedings of the 6th International Conference on Autonomic Computing*. ACM, 13–22.
[30] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.
[31] JustGlowing. [n.d.]. MiniSom: a minimalistic implementation of the Self Organizing Maps. Retrieved from https://github.com/JustGlowing/minisom.
[32] Eamonn Keogh, Jessica Lin, and Ada Fu. 2005. Hot sax: Efficiently finding the most unusual time series subsequence. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM'05)*. Ieee, 8–pp.
[33] Keras. [n.d.]. Keras: The Python Deep Learning library. Retrieved from https://keras.io/.
[34] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. Retrieved from https://arXiv:1412.6980.
[35] Mayuresh Kunjir, Yuzhang Han, and Shivnath Babu. 2016. Where does memory go?: Study of memory management in JVM-based data analytics.
[36] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM'07)*. IEEE, 583–588.
[37] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering-based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 102–111.
[38] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *Proc. VLDB Endow.* 12, 4 (2018), 390–403.
[39] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection. In *Proceedings of the USENIX Annual Technical Conference*. 1–14.
[40] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2012. A lightweight algorithm for message type extraction in system application logs. *IEEE Trans. Knowl. Data Eng.* 24, 11 (2012), 1921–1936.
[41] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. 2016. LSTM-based encoder-decoder for multi-sensor anomaly detection. Retrieved from https://arXiv:1607.00148.
[42] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. 2015. Long short term memory networks for anomaly detection in time series. In *Proceedings*. Presses Universitaires de Louvain, 89.
[43] Shigeru Maya, Ken Ueno, and Takeichiro Nishikawa. 2019. dLSTM: A new approach for anomaly detection using deep learning with delayed prediction. *Int. J. Data Sci. Anal.* (2019), 1–28.
[44] Mohammad A. Munawar and Paul A. S. Ward. 2007. A comparative study of pairwise regression techniques for problem determination. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 152–166.
[45] Mohammad Ahmad Munawar and Paul A. S. Ward. 2007. Leveraging many simple statistical models to adaptively monitor software systems. In *Proceedings of the International Symposium on Parallel and Distributed Processing and Applications*. Springer, 457–470.
[46] Openjdk. [n.d.]. Openjdk documentation. Retrieved December 2, 2019 from http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html
[47] Oracle. [n.d.]. jconsole. Retrieved from http://openjdk.java.net/tools/svc/jconsole/.

[48] Rob Powers, Moises Goldszmidt, and Ira Cohen. 2005. Short term performance forecasting in enterprise systems. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. ACM, 801–807.

[49] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *Proceedings of the IEEE 31st International Conference on Data Engineering*. IEEE, 1167–1178.

[50] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. 2014. An empirical study of long lived bugs. In *Proceedings of the Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*. IEEE, 144–153.

[51] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. 2009. Reference-driven performance anomaly identification. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 37. ACM, 85–96.

[52] solarwinds. [n.d.]. Solarwinds SAM Server & Application Monitor. Retrieved from https://www.solarwinds.com/server-application-monitor?CMP=BIZ-TAD-PCWDLD-SAM_PP-A-PP-Q116.

[53] Christopher Stewart, Terence Kelly, and Alex Zhang. 2007. Exploiting nonstationarity for performance prediction. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 31–44.

[54] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Proceedings of the 13th Annual Conference of the International Speech Communication Association*.

[55] Yongmin Tan and Xiaohui Gu. 2010. On predictability of system anomalies in real world. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 133–140.

[56] Yongmin Tan, Xiaohui Gu, and Haixun Wang. 2010. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 173–182.

[57] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 285–294.

[58] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. ACM, 785–794.

[59] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. 2016. Anomaly detection in automobile control network data with long short-term memory networks. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA'16)*. IEEE, 130–139.

[60] Avishay Traeger, Ivan Deras, and Erez Zadok. 2008. DARC: Dynamic analysis of root causes of latency distributions. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 277–288.

[61] Dylan Tweney. 2013. Amazon website goes down for 40 minutes, costing the company $5 million. Retrieved from https://venturebeat.com/2013/08/19/amazon-website-down/.

[62] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM'03)*. IEEE, 119–126.

[63] VMware. [n.d.]. Virtual machine CPU usage alarm. Retrieved from https://kb.vmware.com/s/article/2057830.

[64] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. 2010. Online detection of utility cloud anomalies using metric distributions. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS'10)*. IEEE, 96–103.

[65] James C. Warner. 2013. top, Linux man page. Retrieved from https://linux.die.net/man/1/top.

[66] Andrew W. Williams, Soila M. Pertet, and Priya Narasimhan. 2007. Tiresias: Black-box failure prediction in distributed systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[67] Cort J. Willmott and Kenji Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Res.* 30, 1 (2005), 79–82.

[68] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online system problem detection by mining patterns of console logs. In *Proceedings of the 9th IEEE International Conference on Data Mining*. IEEE, 588–597.

[69] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 117–132.

[70] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 199–208.

[71] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau. 2015. A C-LSTM neural network for text classification. Retrieved from https://arXiv:1511.08630.

[72] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2018. Tools and benchmarks for automated log parsing. Retrieved from https://arXiv:1811.03509.

Received March 2020; revised November 2020; accepted November 2020