2022-09

ALBADross: active learning based anomaly diagnosis for production HPC systems

ALBADross: Active Learning Based Anomaly Diagnosis for Production HPC Systems

Burak Aksar*, Efe Sencan*, Benjamin Schwaller†, Omar Aaziz†,
Vitus J. Leung†, Jim Brandt†, Brian Kulis*, Ayse K. Coskun*
*Electrical and Computer Engineering, Boston University, Boston MA 02215, USA
{baksar, esencan, bkulis, acoskun}@bu.edu
†Sandia National Laboratories, Albuquerque NM 87123, USA
{bschwal, oaaziz, vjleung, brandt}@sandia.gov

***Abstract*—Diagnosing causes of performance variations in High-Performance Computing (HPC) systems is a daunting challenge due to the systems' scale and complexity. Variations in application performance result in premature job termination, lower energy efficiency, or wasted computing resources. One potential solution is manual root-cause analysis based on system telemetry data. However, this approach has become an increasingly time-consuming procedure as the process relies on human expertise and the size of telemetry data is voluminous. Recent research employs supervised machine learning (ML) models to diagnose previously encountered performance anomalies in compute nodes automatically. However, these models generally necessitate vast amounts of labeled samples that represent anomalous and healthy states of an application during training. The demand for labeled samples is constraining because gathering labeled samples is difficult and costly, especially considering anomalies that occur infrequently. This paper proposes a novel active learning-based framework that diagnoses previously encountered performance anomalies in HPC systems using significantly fewer labeled samples compared to state-of-the-art ML-based frameworks. Our framework combines an active learning-based query strategy and a supervised classifier to minimize the number of labeled samples required to achieve a target performance score. We evaluate our framework on a production HPC system and a testbed HPC cluster using real and proxy applications. We show that our framework, *ALBADross*, achieves a 0.95 F1-score using 28x fewer labeled samples compared to a supervised approach with equal F1-score, even when there are previously unseen applications and application inputs in the test dataset.**

***Index Terms*—anomaly diagnosis, performance, active learning**

I. Introduction

HIGH-performance computing (HPC) systems provide significant computing resources for a wide range of scientific and engineering applications. Generally, HPC systems comprise thousands of compute nodes to provide higher degrees of resource sharing at scale. With the emerging exascale machines, the underlying infrastructure has become more heterogeneous and complex [1]. The ever-growing complexity of the underlying infrastructure is more prone to a substantial increase in performance variation and degradation, such as up to 8 times delay in the job execution time and more than 70% variation in application performance with the same input deck [2]–[4]. Performance variations can be caused by performance anomalies such as hardware-related problems [5], shared resource contention [6], [7], fluctuating CPU frequency [8], orphan processes [9], and memory-related problems (e.g., memory leak) [10].

Most HPC systems include monitoring and logging frameworks that allow for collecting telemetry data in the form of logs, traces, and performance metrics. One way to assess system health is by investigating selected performance metrics and the logs of specific subsystems. Selected performance metrics and log messages may shed light on the root causes of performance anomalies; however, considering billions of data points are generated daily [11], manual analysis is not feasible and highly error-prone. Instead of manual analysis, it is possible to establish rule-based heuristics to check abnormal log sequences and performance metrics deviating from the average values [12]–[15]. However, again, these methods are fragile, have limited scalability, and are insufficient to handle enormous volumes of telemetry data. Given the limits of manual analysis, machine learning (ML) is emerging as an attractive approach for automating performance analytics. ML-based telemetry data analytics are promising since they reduce diagnostics time and help rapid mitigation of performance problems. In this paper, we specifically focus on *diagnosing* performance anomalies (i.e., providing the type of the detected anomaly) on HPC systems instead of anomaly detection, which only informs whether an application run is anomalous.

To help identify performance anomalies in HPC systems, several ML-based frameworks that detect and diagnose performance anomalies have been introduced. In this work, we are interested in supervised and semi-supervised ML-based frameworks because we assume at least a few *labeled* samples exist (i.e., labeled as healthy or anomalous, with the specific type of anomaly). A *sample* refers to the whole set of telemetry data collected during the execution of an application on a compute node. Supervised frameworks that use Support Vector Machines (SVM) [16], k-nearest neighbors [17], random forest [18], [19], or a Bayesian classifier [20] have been shown to successfully perform anomaly detection and/or diagnosis when enough labeled samples exist. A notable downside of such fully supervised frameworks is the vast amount of labeled samples required to represent the healthy and anomalous states of an application during training. In production HPC systems, plenty of unlabeled samples are typically available,

but obtaining a large number of labeled samples, especially for anomalies, is a complex and time-consuming task for a human annotator (e.g., system administrator or application developer). Hence, the required number of labeled samples should be kept at a minimum. It is also impractical to expect access to labeled samples for all kinds of different applications and anomalies, as the number of combinations can exceed hundreds, even without considering different application input decks. In a realistic setting, a framework should be robust[1] against previously unseen applications and application inputs. We observe that the diagnosis performance of a state-of-the-art supervised framework reduces by up to 30% when the test dataset contains previously unseen applications, so such frameworks may not easily fit a production system setting due to limitations in robustness.

One approach to minimize the number of labeled samples is applying semi-supervised techniques that leverage both labeled and unlabeled samples. For example, some semi-supervised frameworks focus on detecting anomalies using only labeled healthy samples [21], [22]. Even though anomaly detection can help, identifying the root cause of performance anomalies may significantly improve the system performance and lead to an effective mitigation. Aksar et al. design a semi-supervised framework, which operates with a large number of unlabeled and a few labeled samples to diagnose performance anomalies in production HPC systems [23]. We also assume a similar setting where only a few labeled samples exist from each class, whereas the number of unlabeled samples is large. In addition, our setting assumes that a human annotator is available to provide the label of a selected sample upon request.

This paper proposes a novel **A**ctive **L**earning-**B**ased **A**nomaly **D**iagnosis (*ALBADross*) framework, which uses significantly fewer labeled samples compared to existing ML-based frameworks, while providing robustness for previously unseen applications and application inputs. *ALBADross* utilizes resource usage characteristics of applications collected by monitoring frameworks to train an ML model with a single sample for each application and anomaly pair. Then, by employing active learning-based query strategies, we determine which samples should be labeled among thousands of unlabeled samples in order to reach a target anomaly diagnosis score. We evaluate *ALBADross* on a production HPC system and a testbed HPC cluster using real applications and benchmark suites with synthetic anomalies. Our specific contributions are as follows:

- Design of an active learning-based anomaly diagnosis framework, *ALBADross*, that operates with a small number of informative samples to reach a target anomaly diagnosis score[2].
- Demonstration of *ALBADross* using real applications on a production HPC system. *ALBADross* achieves a 0.95 F1-score and near-zero false alarm rate, while using 28x fewer labeled samples compared to a fully-supervised solution with equal accuracy, even when previously unseen applications and application inputs exist in the test dataset.
- Investigation and quantification of the *robustness* of *ALBADross* and other ML-based methods under various production system scenarios.

The rest of the paper starts with an overview of the related work. Sec. III describes the technical details of the proposed framework, Sec. IV explains our experimental methodology, Sec. V presents our results, and we conclude in Sec. VI.

II. Related Work and Background

Performance variation has been a significant area of research for large-scale computing systems, and it will continue to be a formidable problem, even more so as we approach the exascale computing era. This section summarizes recent research on ML-based telemetry data analytics in two main subsections.

A. Active Learning and Anomaly Detection

Active learning is a unique form of semi-supervised learning based on querying samples iteratively from an unlabeled dataset and re-training the selected model with the new labels learned during the query process [24]. The primary motivation behind active learning is that an ML model can achieve better prediction accuracy if it can pick the samples to be labeled for the training stage. Therefore, active learning, in its basis, assumes that there is a human annotator who can provide labels of the selected samples. Active learning assumes three distinct scenarios: membership query synthesis [25], stream-based selective sampling [26], and pool-based sampling [27]. In the membership query synthesis, the active learner can generate its own instances to increase the number of labeled samples based on the underlying data distribution. In stream-based selective sampling, the samples from the unlabeled dataset are shown to the active learner one by one, and the learner decides whether or not to learn the label of the sample according to a pre-defined uncertainty threshold. The pool-based sampling assumes that the active learner has access to a large unlabeled dataset. The learner then determines the samples to be labeled based on the selected query strategy. Since unlabeled samples are voluminous and accessible at once, pool-based sampling is the most suitable scenario for production HPC systems. Various query strategies can be utilized in the pool-based sampling (see Sec. III).

Various studies leverage active learning for anomaly detection or increasing the prediction performance when the labeled data is limited. Russo et al. use a random forest model and classification uncertainty query strategy to detect anomalous points from environmental sensor data [28]. Chen et al. diagnose faults in mechanical gearbox systems by employing active learning with the classification uncertainty query strategy [29]. Another study first extracts feature vectors from raw signals using singular value decomposition and then utilizes random forest as a supervised classifier for the

[1] In this paper, we refer to *robustness* as being able to detect and diagnose performance anomalies with high accuracy when previously unseen applications and application inputs exist in the test dataset.

[2] Our implementation is available at: github.com/peaclab/ALBADross

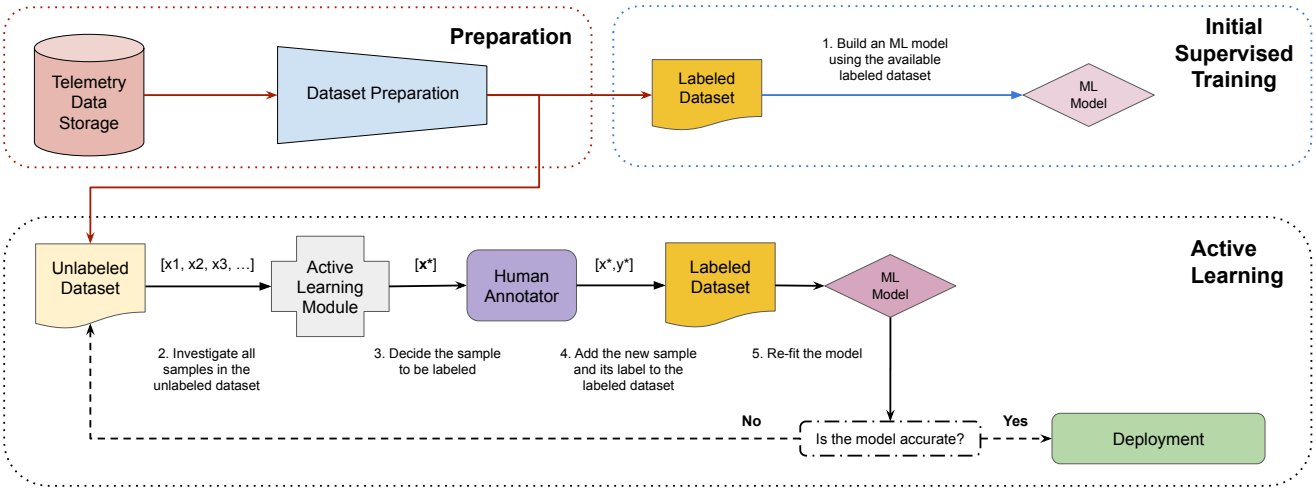

Fig. 1. The high-level architecture of *ALBADross*. First, we apply feature extraction and feature selection to the available telemetry data. Next, we train a supervised ML model using the available labeled samples. In the second step, the active learning module investigates the available unlabeled samples and determines which sample should be labeled based on the selected query strategy. Then, a human annotator provides the label, y^*, for the selected sample, x^*. In the last stage, the model is re-trained with the updated labeled dataset. We repeat the same cycle starting from the second step until we achieve the target performance score, at which point the model can be deployed.

diagnosis stage [30]. Wang et al. propose an active learning-based robust random cut forest algorithm [31] for detecting anomalies in key performance indicator time series data [27]. One method involves a semi-supervised anomaly detection framework for time series data that combines deep reinforcement learning and active learning [32]. Xie et al. combine active learning with one-class SVM to detect anomalous runtime behaviors in HPC applications [33]. The authors model instructions of executed applications as call stack trees, which include temporal and contextual information, and aim to find anomalous tree structures. Pimentel et al. utilize active learning along with denoising autoencoder and multi-layer perceptron to detect anomalies in an unsupervised manner [34]. Bodor et al. propose an active learning-based isolation forest model to detect anomalies in multivariate KPI time series data [35]. Nixon et al. combine active learning with autoencoder to detect anomalies in the network data stream [36].

B. Machine Learning for HPC Performance Analytics

Supervised and semi-supervised ML frameworks, which operate using multivariate time series telemetry data, are widely employed to accomplish various tasks such as performance anomaly diagnosis and application detection. In the supervised setting, an ML model has access to all ground truth labels during training. Tuncer et al. propose a novel supervised anomaly diagnosis framework, which applies statistical feature extraction along with a feature selection technique, and then trains a decision tree-based classifier to classify the type of an anomaly [37]. Ates et al. classify running applications on compute nodes and detect unknown applications using a supervised ML model [38]. Baseman et al. propose a framework that leverages kernel density estimation to generate synthetic samples and apply a random forest classifier to predict performance anomalies [39]. In the semi-supervised setting, the model has access to a few labeled and a large set of unlabeled samples. Borghesi et al. use an autoencoder to learn the normal behavior of compute nodes and detect anomalies based on the reconstruction error [22]. Aksar et al. propose an autoencoder-based semi-supervised framework to diagnose anomalies in production HPC systems [23]. Their framework learns the characteristics of previously encountered performance anomalies in an unsupervised fashion and then leverages supervised classifiers to diagnose anomalies on compute nodes.

The key differentiators of *ALBADross* are two-fold. Existing semi-supervised frameworks [21]–[23] aim to detect and diagnose performance anomalies using the existing labeled and unlabeled samples, whereas *ALBADross* aims to minimize the labeled samples required for a target diagnosis score in the presence of a human annotator. The closest related work, which combines active learning and supervised classification in the HPC domain, only detects anomalous executions in applications [33]. However, *ALBADross* is the first work, to the best of our knowledge, that combines active learning and supervised classification to diagnose performance anomalies with only a few labeled samples.

III. Our Proposed Framework: *ALBADross*

Our primary goal is to determine whether an application run in a system displays anomalous behavior (i.e., performance variability) and, if so, to characterize the reason behind the anomalous behavior (e.g., memory leak, I/O contention, etc.) in an application-agnostic manner. We are particularly interested in anomalies that result in performance variability, where applications continue to run without terminating/crashing. Such anomalies are typically more challenging

to discover and diagnose than failures that result in software errors or premature termination.

We propose an active learning-based anomaly diagnosis framework called *ALBADross*, which achieves a target anomaly diagnosis score while minimizing the number of samples that need to be labeled. Figure 1 shows an overview of our framework. We collect telemetry data from compute nodes while running applications with and without (synthetic) anomalies. We then extract statistical features from the raw time series and train a supervised model with one sample for each application and anomaly pair. After the initial model is trained, the active learning module investigates samples in the unlabeled dataset and selects samples by utilizing different query strategies. Then, a human annotator provides labels for the selected samples. As a final stage, we re-train the model including the newly labeled samples. Note that our system is independent of the monitoring framework used. In the following sections, we explain these processes in depth.

A. Statistical Feature Extraction

The main goal of this stage is to extract useful information from raw telemetry time series data. We use two different open-source feature extraction toolkits. The first one is *MVTS* [40], which computes 48 statistical features for every metric. Some features are descriptive statistics (e.g., min, max, mean, etc.), absolute differences between the descriptive statistics of the first and second halves of the time series, and long-run trends (e.g., longest monotonic increase). The second one is *TSFRESH* [41], which computes 794 features for every metric based on 63 different time series characterization methods. TSFRESH extracts a more extensive and more advanced set of features compared to MVTS, including, but not limited to, approximate entropy [42], power spectral density of time series [43], and variation coefficient of each feature.

B. Feature Selection

After extracting statistical features, we use the Chi-Square feature selection [44] technique to reduce the data dimensionality, which is often helpful in time series regression and classification tasks [45]. In statistics, the Chi-Square test is used to determine the independence of two events. The goal is to select features that are significantly dependent on the given output. When two features are independent, the observed count approaches the expected value, resulting in a lower Chi-Square score. Thus, a high Chi-Square score suggests that the independence hypothesis is false. In simple terms, the higher the Chi-Square score, the more reliant the feature is on the output and hence more suitable for model training. In our experiments, we compute a Chi-Square score for every feature, sort them in descending order, and select the top-k features. The exact implementation details are disclosed in Sec. IV.

C. Hyperparameter Search and Supervised Training

This stage aims to identify the optimal hyperparameters for each supervised model before applying active learning. We tune each model's hyperparameters using grid search across multiple cross-validation folds. For the initial model training stage, we assume one labeled sample from each application and anomaly pair exist to train a supervised ML model with the selected hyperparameters.

D. Active Learning

After the initial model training is completed with the selected hyperparameters, we apply active learning to determine which samples should be labeled. In the context of active learning, we assume a pool-based sampling scenario where a large number of unlabeled samples are available for the query process. We explore three different query strategies to determine samples to be labeled, where each strategy tries to select a sample to be labeled by leveraging different statistical measurements.

1) Classification Uncertainty: One of the commonly used strategies is classification uncertainty, which is defined as follows:

$$U(x) = 1 - P(y|x), \tag{1}$$

where x is the instance to be predicted and y is the most likely prediction. We represent class probabilities with $\hat{p_i} = [p_1, p_2, p_3, ..., p_k]$, where i denotes the sample's index, p_k is the probability for the k-th class, and $\hat{p_i}$ contains all class probabilities. Assume we have the following class probabilities for three different samples:

$$\begin{aligned} \hat{p_1} &= [0.1, 0.85, 0.05] \\ \hat{p_2} &= [0.6, 0.3, 0.1] \\ \hat{p_3} &= [0.39, 0.61, 0.0] \end{aligned} \tag{2}$$

The active learner sequentially queries all the unlabeled instances until it finds the sample for which its current prediction is maximally uncertain. In the given example, uncertainties are $U_{list} = [0.15, 0.4, 0.39]$, and the selected sample is the second one in this scenario.

2) Classification Margin: The classification margin strategy calculates the probability difference between the first and second most likely predictions, which is defined as follows:

$$M(x) = P(y_1|x) - P(y_2|x), \tag{3}$$

where y_1 and y_1 are the first and second most likely classes. Using the given class probabilities for each sample in Eq. 2, the margins are the following: $M_{list} = [0.75, 0.3, 0.22]$. When deciding on the sample to be labeled, the strategy chooses the sample with the smallest margin, as a smaller margin indicates a more uncertain decision. It is the third sample in this scenario.

3) Classification Entropy: The classification entropy is proportional to the average number of guesses one has to make to find the true class, which is defined as follows:

$$H(x) = -\sum_k p_k \log(p_k). \tag{4}$$

Using the example class probabilities, the calculated entropies are the following: $H_{list} = [0.52, 0.90, 0.67]$. This strategy

TABLE I
APPLICATIONS WE RUN ON VOLTA FOR DATA COLLECTION.

Benchmark	Application	Description
NAS	BT	Block tri-diagonal solver
	CG	Conjugate gradient
	FT	3D Fast Fourier Transform
	LU	Gauss-Seidel solver
	MG	Multi-grid on meshes
	SP	Scalar penta-diagonal solver
Mantevo	MINIMD	Molecular dynamics
	COMD	Molecular dynamics
	MINIGHOST	Partial differential equations
	MINIAMR	Stencil calculation
Other	KRIPKE	Particle transport

selects the instance where the class probabilities have the highest entropy, which is the first sample in this case. After the selected query strategy determines the sample to be labeled, we re-train the model including the newly labeled sample instead of training it from scratch.

E. Anomaly Detection and Diagnosis

We monitor the training process and finalize the training when we reach the maximum number of allowed queries or the target diagnosis score. The final model is stored as a *pickle* object, and for a given sample, it returns the diagnosed anomaly label and its confidence.

IV. EXPERIMENTAL METHODOLOGY

The first section details the HPC systems and the applications we run across these systems. Following that, we discuss the details of the monitoring framework and synthetic anomalies that we employ to generate performance variation. We conclude with implementation details.

A. HPC Systems and Applications

We conduct tests on a testbed system called Volta and on a production HPC system called Eclipse. We test our framework's performance against baselines using both benchmarks and real applications.

(1) Volta is a Sandia National Laboratories-based Cray XC30m testbed supercomputer. Volta comprises 52 computing nodes organized in 13 connected switches, each with four nodes. Each node features 64GB of memory and two sockets, each of which is equipped with an Intel Xeon E5-2695 v2 CPU featuring 12 two-way hyper-threaded cores. In Volta, we employ NAS Parallel Benchmarks (NPB) [46] and Mantevo Benchmark Suite [47], developed for performance and scalability research, to cover a broad collection of HPC applications. In addition, we use the *Kripke* application, a proxy application that mimics particle transit [48]. Table I contains a list of all applications used in our experiments. We run each program for 10-15 minutes over 4 compute nodes with three inputs for each application.

(2) Eclipse is a production HPC system located at Sandia National Laboratories. Eclipse has 1488 compute nodes and a peak performance of 1.8 petaflops. Each node comes equipped with 128GB of RAM and two sockets, each of which houses 18 E5-2695 v4 CPU cores. We run the following applications on Eclipse: *LAMMPS*, *HACC*, *sw4*, *ExaMiniMD*, *SWFFT*, and *sw4lite*. There are three real applications among them: *LAMMPS*, a molecular dynamics simulation focusing on materials modeling [49]; *HACC*, an extreme-scale cosmological simulation [50]; and *sw4*, a popular 3D seismic model [51]. The remaining three, *ExaMiniMD*, *SWFFT*, and *sw4lite*, are proxy applications from the ECP Proxy Apps Suite [52]. We list all applications used in our experiments in Table II. We run each application on 4, 8, and 16 nodes, where each application has a different input per unique node count, for 20-45 minutes.

TABLE II
APPLICATIONS WE RUN ON ECLIPSE FOR DATA COLLECTION.

	Application	Description
Real Applications	LAMMPS	Molecular dynamics
	HACC	Cosmological simulation
	SW4	Seismic modeling
ECP Proxy Suite	EXAMINIMD	Molecular dynamics
	SWFFT	3D Fast Fourier Transform
	SW4LITE	Numerical kernel optimizations

TABLE III
A LIST OF THE HPAS ANOMALIES USED IN OUR EXPERIMENTS.

Anomaly type	Anomaly behavior
CPU intensive process	Arithmetic operations
Cache contention	Cache read & write
Memory bandwidth contention	Uncached memory write
Memory leakage	Increasingly allocate & fill memory

B. Monitoring Framework

We collect telemetry data across multiple subsystems using the Lightweight Distributed Metric Service (LDMS) [11]. LDMS can gather thousands of distinct resource utilization metrics and performance counters at sub-second resolution from compute nodes. We gather 806 metrics from Eclipse and 721 metrics from Volta at a rate of 1Hz. Some example metrics from each subsystem are listed below:

- Memory and virtual memory (e.g., free, active memory)
- CPU (e.g., per-core system, user, and idle time)
- Network (e.g., number of received/transmitted packets)
- Shared file system (e.g., open, close, and write counts)
- Cray performance counters (e.g., power consumption, write-back counters)

C. Synthetic Performance Anomalies

ML models must see samples exhibiting anomalous properties to learn and detect specific anomaly signatures; on the other hand, performance anomalies are rare, especially in production HPC systems. Hence, we utilize synthetic performance anomalies from the open-source HPC Performance Anomaly Suite (HPAS) [53] to systematically simulate anomalous application behavior.

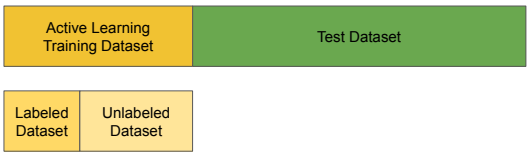

Fig. 2. Splitting the dataset into training and test datasets. The labeled dataset represents the initial condition for the supervised training phase, where one sample from each application and anomaly pair exists. Query strategies and baselines use the unlabeled dataset to determine samples to be labeled.

HPAS is a free and open-source performance anomaly suite that can be used to replicate the most popular performance anomalies. In HPAS, synthetic anomalies target five primary subsystems: the CPU, the cache, the memory, the network, and shared storage. We inject anomalies using a variety of settings to simulate various levels of performance variance, as detailed in Table III. While running a multi-node application, we run a synthetic anomaly in the first allocated compute node. For Volta dataset, six different anomaly intensities are used: 2%, 5%, 10%, 20%, 50%, and 100%. For the Eclipse dataset, we have 2 or 3 settings including different intensities for each anomaly type. If an anomaly is injected, the data for each compute node is labeled with an anomaly type; otherwise, the sample is labeled as *healthy*.

D. Baselines

The most common baseline in the active learning domain is the random selection, referred to as *Random* [54]. In each iteration, we randomly select a sample from the unlabeled dataset and ask for its label. Even though *Random* is pretty typical in the literature, we implement another baseline scenario, assuming that the applications running on the system are known. In each iteration, we select one sample from each application type, referred to as *Equal App*. For example, we query in increments of 6 and guarantee that there is one sample from each application type since the Eclipse dataset has six applications in total. The last baseline we implement is *Proctor*, which is an autoencoder-based semi-supervised ML framework to diagnose performance anomalies [23]. In each iteration, we query new samples using *Random* and retrain the supervised classifier. We provide the implementation details in the next section.

E. Implementation Details

1) Data Collection & Preparation: For every application that is run on a compute node, we obtain a sequence of multivariate time series data R^{TxM} where T is the number of timestamps, and M is the number of metrics of the multivariate time series data. First, we omit the time intervals corresponding to the initialization and termination phases since some metrics may fluctuate significantly from their expected values. Then, we calculate the difference between each step for cumulative performance counters, as we are interested in the change, not the raw value. Finally, we perform linear interpolation in every time series to fill in missing values since some metric values may get lost during data collection.

TABLE IV
HYPERPARAMETER SEARCH SPACE FOR EACH MODEL AND THEIR OPTIMAL HYPERPARAMETERS FOR ECLIPSE* AND VOLTA+ DATASETS

Model	Hyperparameter Space
LR	penalty: $l1^{*+}$, l2
	C: 0.001, 0.01, 0.1, 1.0^{*}, 10.0^{+}
RF	n_estimators: 8, 10, 20^{+}, 100, 200^{*}
	max_depth: None, 4, 8^{*+}, 10, 20
	criterion: gini, $entropy^{*+}$
LGBM	num_leaves: 2, 8, 31, 128^{+}
	learning_rate: 0.01, 0.1^{*+}, 0.3
	max_depth: -1^{*}, 2, 8^{+}
	colsample_bytree: 0.5, 1.0^{*+}
MLP	max_iter: 100^{*+}, 200, 500, 1000
	hidden_layer_sizes: (10,10,10), $(50, 100, 50)^{*}$, $(100)^{+}$
	alpha: 0.0001^{*}, 0.001, 0.01^{+}

After the data cleanup, we extract statistical features of raw time series using MVTS and TSFRESH feature extraction toolkits and drop features with NaN or zero values. After this step, we obtain 6436 and 80839 features in the Eclipse dataset and 5445 and 99169 features in the Volta dataset, with MVTS and TSFRESH, respectively. Using the Chi-Square feature selection technique, we experiment with the different number of features: 250, 500, 1000, 2000, 4000, 6436. We limit the minimum to 250 features since we observe a decreasing trend in the prediction performance. The maximum number of features is 6436 because this is the maximum number of features for the Eclipse dataset. We achieve the best F1-score with the following combinations, MVTS - 2000 features, and TSFRESH - 2000 features, in the Eclipse and the Volta datasets, respectively.

2) Dataset Split & Hyperparameter Tuning: Figure 2 shows the dataset split to mimic a real production system scenario during our experiments. The active learning training dataset comprises labeled and unlabeled datasets. The labeled dataset is used to train supervised models before applying active learning. For the Eclipse and Volta datasets, it contains 30 (i.e., six applications and five anomalies) and 55 samples (i.e., 11 applications and five anomalies), respectively. We repeat the train-test split process five times using stratified sampling to ensure that the class distribution matches the distribution in the entire dataset. We apply the *Min-Max* scaler to training and test datasets. We also maintain a 10% anomaly ratio (i.e., the number of anomalous samples divided by all samples) in the active learning training dataset. To make sure this ratio aligns with a production system setting, we investigate the percentage of application runs (without injecting any synthetic anomalies) that show an outlier behavior (i.e., runs that have execution time 1.5 IQR below Q1 or 1.5 IQR above Q3) in terms of execution time on Eclipse. We observe an outlier ratio between

TABLE V
THE SUMMARY OF OUR ANOMALY DIAGNOSIS RESULTS. FOR EACH DATASET, WE REPORT THE REQUIRED NUMBER OF SAMPLES TO ACHIEVE A CERTAIN F1-SCORE WITH THE BEST FEATURE EXTRACTION METHOD AND QUERY STRATEGY.

Dataset	Feature Extraction Method	Query Strategy	Initial Sample Count	Starting F1-score	F1-score:0.85	F1-score:0.90	F1-score:0.95	Active Learning Training Dataset F1-score	Max Score 5-fold CV
Volta	TSFRESH	Uncertainty	55	0.86	Already Passed	65 Samples	76 Samples	0.95 (6329 Samples)	0.99 (16732 Samples)
Eclipse	MVTS	Margin	30	0.72	87 Samples	150 Samples	230 Samples	0.95 (5619 Samples)	0.99 (19652 Samples)

2-7% among different application types and we cap this value to 10%. The size of Eclipse and Volta datasets is different; hence, training and test percentages are adjusted accordingly for each dataset to satisfy the conditions mentioned above.

In the last stage, we perform hyperparameter tuning only on the active learning training dataset to prevent potential information leakage from the test set, i.e., the test dataset is withheld. We apply grid search in a 5-fold stratified cross-validation setting. The hyperparameter search space for each model is shown in Table IV. The optimal hyperparameters are denoted with the * and $^+$ symbols for Eclipse and Volta datasets, respectively.

3) ML Models and Baselines: We use the random forest, Light Gradient Boosting Machines (LGBM) [55], logistic regression, and Multi-Layer Perceptron (MLP) in our experiments. A random forest model comprises numerous decision trees, and it often uses the average of each decision tree or majority voting to make predictions. Gradient boosting machines are decision-tree-based classifiers that make use of gradient boosting, which generates a prediction result by combining weak prediction models. We use the LGBM since they generally require less time to train and have lower memory usage compared to Extreme Gradient Boosting, which is another popular gradient boosting machine algorithm [56]. We utilize the scikit-learn [57] and the modAL [58] libraries to implement ML models and active learning query strategies, respectively. Each of the baseline methods, *Random* and *Equal App*, is run ten times in every query. We adopt the proposed autoencoder topology to implement *Proctor* [23]. The final model includes a deep autoencoder with 2000 neurons in the code layer and a logistic regression classifier for the supervised training part. We use *adadelta* optimizer and minimize the *Mean Squared Error* for 100 epochs.

V. EVALUATION

We evaluate our framework against three distinct experimental scenarios and present F1-scores, false alarm rates (i.e., false-positive rates), and anomaly miss rates (i.e., false-negative rates). The F1-score is the harmonic mean of precision and recall, where precision shows what percentage of positive class predictions are correct and recall shows what percentage of actual positive class samples are identified correctly. The false alarm rate is the percentage of healthy samples classified as one of the anomaly classes. The anomaly miss rate is the percentage of anomalous samples (any anomaly) that are classified as healthy.

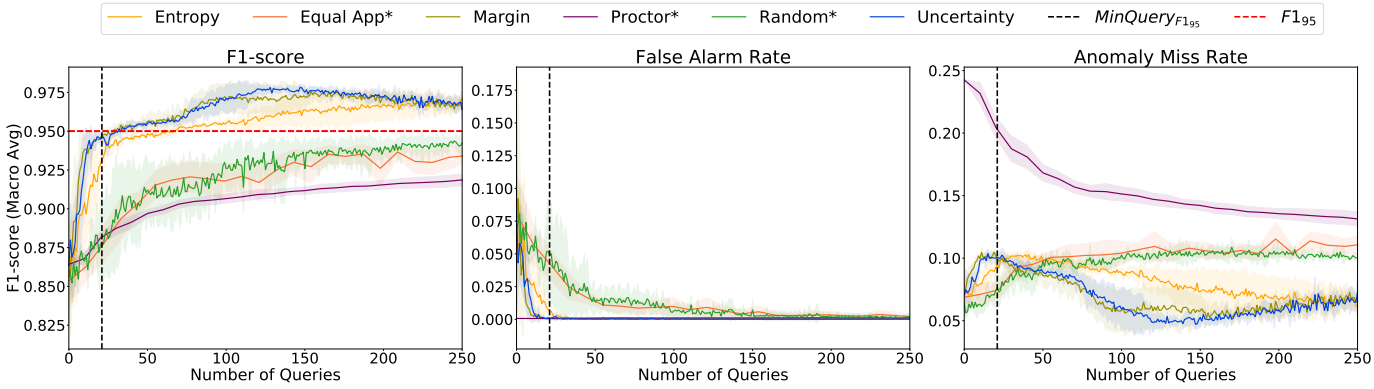

Fig. 3. The F1-scores, false alarm rates, and anomaly miss rates of different query strategies and baselines (*Random**, *Equal App**, and *Proctor**) for the first 250 queries in the Volta dataset. The red dashed line points to a 0.95 F1-score, the black dashed line shows the minimum number of additional samples to reach a 0.95 F-1 score, and the shades show a 95% confidence interval for different train-test splits. The uncertainty query strategy achieves a 0.95 F1-score by learning the labels of additional 21 samples while maintaining an almost perfect false alarm rate.

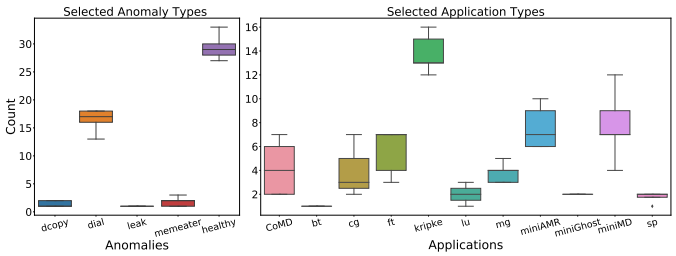

Fig. 4. The distribution of selected application and anomaly types in the Volta dataset for the first 50 queries. The top two labels are *healthy* and *dial*. Among the applications, *Kripke* is the most frequently queried one.

A. Anomaly Diagnosis with Active Learning

The main goal of this scenario is to determine how many labeled samples are required to reach a target F1-score. We explore three different active learning query strategies: entropy, margin, and uncertainty. In terms of baselines, we utilize *Random*, *Equal App*, and *Proctor*. We start with one sample for each application and anomaly pair and query 1000 samples in the Eclipse and Volta datasets for each method. After each query, all the methods are tested with the same test dataset, and then we report the F1-scores, anomaly miss rates, and false alarm rates.

Table V shows the summary of the best results in two datasets. The starting F1-score column shows the initial F1-score when we train the models with the labeled dataset in Fig. 2. In the Volta dataset, the uncertainty query strategy reaches an F1-score of 0.90 with ten additional samples. With an additional 21 samples, the uncertainty query strategy achieves a 0.95 F1-score, whereas *Random* reaches the same F-1 score with approximately 600 samples. This score is the maximum score we can achieve using a random forest model with the active learning training dataset (Fig. 2), which consists of 6329 samples in total. To summarize, the uncertainty query strategy achieves the same score using **only 2%** of the active learning training dataset. For the Eclipse dataset, the margin query strategy can achieve a 0.95 F1-score with additional 200 samples. This score is the maximum score we can achieve using the active learning training dataset (5619 samples), where the margin query strategy achieves this score using **24x fewer** labeled samples.

Figure 3 shows how F1-scores, false alarm rates, and anomaly miss rates change for the first 250 queries of each method in the Volta dataset. Even though margin and uncertainty query strategies perform similarly, the uncertainty query strategy achieves a slightly higher F1-score on average, so it is chosen as the best strategy. Regarding false alarm rates, the uncertainty strategy reaches an almost zero false alarm pretty quickly, whereas the anomaly miss rate increases until the 50^{th} query. To understand the underlying behavior, we investigate the selected anomaly and application types during this process and provide a drill-down analysis for the first 50 queries in Fig. 4. The main reason is that the uncertainty query strategy initially selects more healthy samples, where almost 30 samples out of 50 belong to the *healthy* label on average. This trend leads to an immediate drop in the false alarm rate while creating a temporary increase in the anomaly miss rate since the model prioritizes learning the signatures of healthy samples. Most of the selected healthy samples are from *Kripke*, *MiniMD*, and *MiniAMR* applications. On average, 10 out of 13 *Kripke* samples, 5 out of 8 *MiniMD* samples, and 4 out of 7 *MiniAMR* samples have the *healthy* label. On the other hand, *dial* anomaly is the most queried anomaly type amongst the other anomalies. The reason is that the *dial* anomaly is the most confusing anomaly type for the model since it has the lowest F1-score in the prediction phase. In terms of baselines, *Proctor* has the best false alarm rate but the lowest classification performance. When we investigate F1-scores for each anomaly type, we realize that *Proctor* has over 0.94 F1-score for each class but *cpuoccupy*, which is around 0.73. This trend leads to a decrease in the macro average F1-score. *Proctor* maintains an almost zero false alarm rate from the start, whereas other baselines achieve the same score with 200 additional samples.

Figure 5 shows the anomaly diagnosis results for the Eclipse dataset. The margin query is the best strategy since it achieves

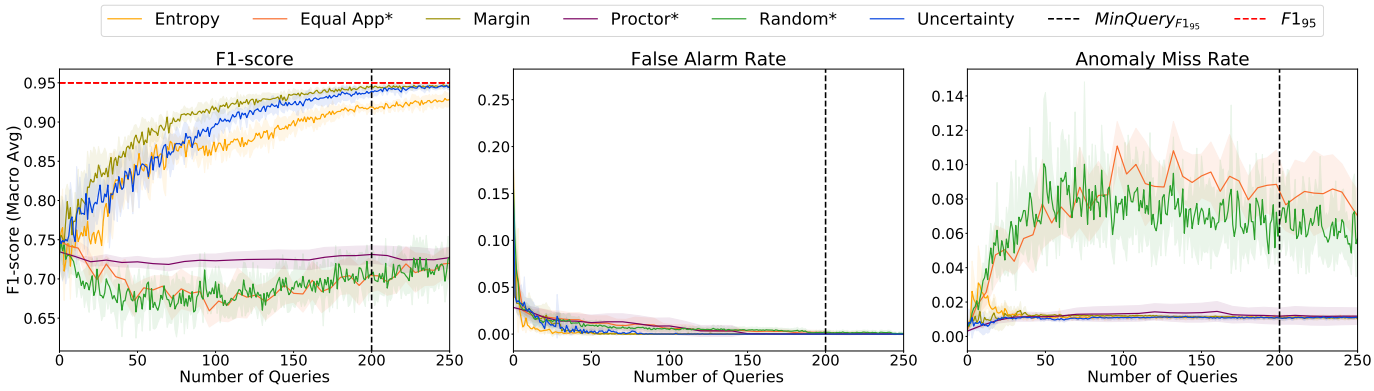

Fig. 5. The F1-scores, false alarm rates, and anomaly miss rates of different query strategies and baselines (*Random**, *Equal App**, and *Proctor**) for the first 250 queries in the Eclipse dataset. The red dashed line points to a 0.95 F1-score, the black dashed line shows the minimum number of additional samples to reach a 0.95 F-1 score, and the shades show a 95% confidence interval across different train-test splits. The margin query strategy achieves a 0.95 F1-score by learning the labels of additional 200 samples while maintaining a zero false alarm rate.

the highest F1-score compared to other query strategies. The margin query strategy reaches a 0.85 F1-score with additional 57 samples, whereas *Random* reaches the same score with approximately 1000 samples. To reach a 0.95 F1-score, the margin query strategy requires approximately 200 samples, which is the same score we can achieve when all 5619 samples in the active learning training dataset are used. One trend we observe is that the number of samples required is almost one order of magnitude higher than the case in the Volta dataset. The main reason behind this trend is the complexity of the dataset. On Eclipse, we run real applications across a different number of compute nodes for 20-45 minutes, whereas on Volta, mostly benchmark and proxy applications are run on four nodes for 10-15 minutes. This complexity also explains the difference between the initial starting F1-scores, which are 0.72 and 0.86 in the Eclipse and the Volta datasets, respectively. Regarding false alarm and anomaly miss rates, while active learning query strategies and *Proctor* perform similarly, *Random* and *Equal App* have high anomaly miss rates. We conduct a similar drill-down analysis to investigate selected applications and anomaly types. The margin query strategy often selects samples with *membw* and *cpuoccupy* since these samples generally have the lowest F1-score. In terms of applications, almost all application types are selected equally. Among the baselines, *Random* has the lowest classification performance, and *Equal App* has the highest anomaly miss rate. The performance of *Proctor* remains steady since the randomly selected labeled samples do not bring extra information to the model.

B. Investigating Robustness for Anomaly Diagnosis

In production systems, it is not likely to obtain labeled samples from every application and their unique inputs. A robust framework should be able to diagnose anomalies even though the test dataset contains previously unseen applications and application inputs. First, we conduct a motivational experiment in the Volta dataset to show the impact of the previously unseen applications on the diagnosis performance. We select three applications for the test dataset and keep the remaining eight in the training dataset. Then, we measure the F1-scores, false alarm rates, and anomaly miss rates in a constant test dataset while gradually increasing the number of applications in the training dataset. This scenario is repeated for different application combinations, i.e., 11 combinations of 3.

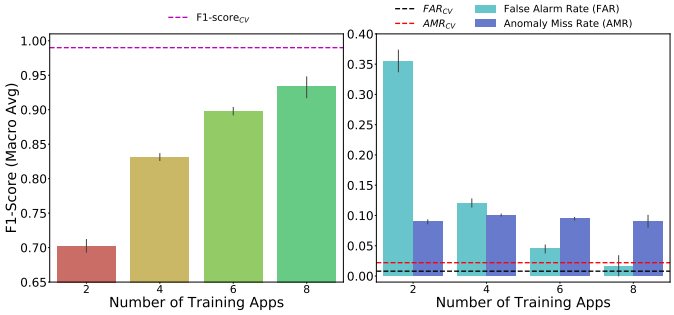

Fig. 7. We increase the number of applications in the training set while keeping a constant test set to measure the robustness of a random forest classifier. The error bars show a 95% confidence interval, and dashed lines show the scores in a 5-fold CV setting when all applications exist in the training and test datasets. When the training dataset has two applications, we achieve a 0.7 average F1-score and 35% false alarm rate, which is 30% lower and 35x higher than the 5-fold CV setting.

Figure 7 summarizes our results. When there are two applications in the training dataset, the average F1-score drops by approximately 30%, and the false alarm rate is 35x higher compared to the 5-fold CV scenario. Even though the average F1-score is 0.7, the F1-score may drop to 0.27 for some application combinations in the training and test datasets.

1) Previously Unseen Applications: Considering the severity of the above problem, we measure the robustness of the best query strategy and *Random* when previously unseen applications exist in the test dataset. We start with two applications in the training dataset (all anomalies are included) and place

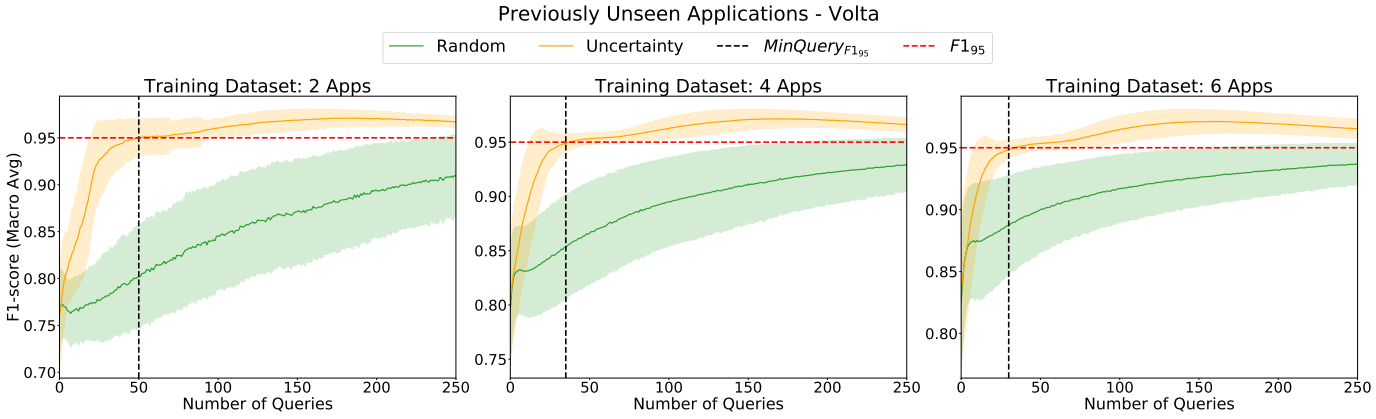

Fig. 6. The F1-scores when the training dataset includes different number of applications. For example, when the training dataset has two applications, the test dataset has the remaining nine applications. The red dashed line points to a 0.95 F1-score, the black dashed line shows the minimum number of additional samples to reach a 0.95 F-1 score, and the shades show one standard deviation for different application combinations. We experiment with all combinations in each scenario (e.g., 11 combinations of 2). When there are two, four, and six applications in the training dataset, the uncertainty query strategy reaches a 0.95 F1-score using additional 50, 35, and 30 samples, respectively.

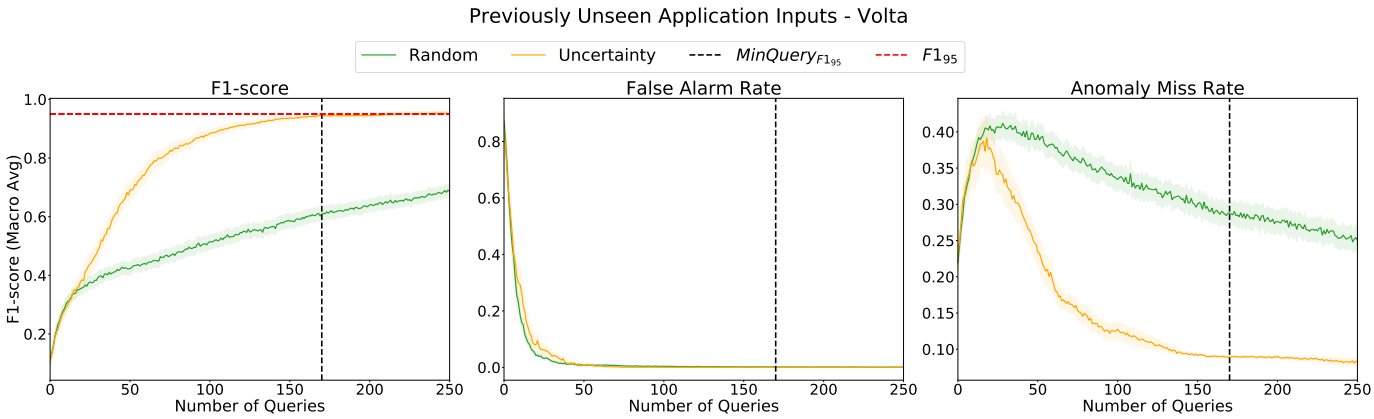

Fig. 8. The F1-scores, false alarm rates, and anomaly miss rates when the training dataset includes only the first input of each application and the test dataset includes the remaining two inputs. The red dashed line points to a 0.95 F1-score, the black dashed line shows the minimum number of additional samples to reach a 0.95 F1-score, and the shades show a 95% confidence interval for different input combinations. The uncertainty query strategy reaches a 0.95 F1-score with 170 additional samples, whereas *Random* requires more than 1000 additional samples.

the remaining applications in the test dataset. In the training dataset, we increase the number of applications to eight with increments of two. Figure 6 shows the F1-scores of three different scenarios, where each scenario has a different number of applications in the training dataset. As we increase the number of applications, we start with a higher initial score and reach a 0.95 F1-score using fewer samples. In all cases, the uncertainty query strategy significantly outperforms *Random* and reaches a 0.95 F1-score by learning the labels of 50 extra samples in the worst case (i.e., the two applications case).

2) Previously Unseen Application Inputs: Another frequent scenario in production systems is executing the same application with different input decks. We repeat a similar scenario to the previously unseen application case, and our goal is to measure the robustness of the best query strategy and *Random*. For each application in our dataset, we have three input decks. First, we delete all runs with the selected input deck from the training dataset and only include the application runs with the deleted input deck to the test dataset. Figure 8 shows F1-scores, false alarm rates, and anomaly miss rates of the uncertainty query strategy and *Random*. The initial F1-score is 0.2, and the initial false alarm rate is 80%, where both scores are significantly worse than the previously unseen application case. This shows that previously unseen application inputs may drastically impact the diagnosis performance. We train a random forest model using the active learning training dataset (6329 samples) and achieve a 0.95 F1-score. To reach the same score, *Random* requires 1000 samples, whereas the uncertainty query strategy requires only 225 samples, which is **28x fewer**. Regarding anomaly miss rate, although we observe a slight increase for the first 20 samples, we see a logarithmic decrease as we continue to select samples to be labeled. The initial increase is due to prioritizing the samples with the *healthy* label.

To summarize, *ALBADross* minimizes the number of samples to be labeled by selecting the most informative samples even though previously unseen applications and application inputs exist, which are very common in a production system.

VI. Conclusion & Future Work

Variation in application performance in HPC systems degrades user satisfaction, reduces resource utilization efficiency, and wastes computing power. With the increasing size and complexity of HPC systems, automated telemetry data-based analytics are becoming essential for reliable and efficient service. Even though active learning-based frameworks have become popular in the domains where the labeled data is limited, none of the prior works leverage it for anomaly diagnosis in HPC. This paper proposes a novel active learning-based framework for diagnosing previously encountered performance anomalies in HPC systems while remaining robust to unseen applications and application inputs. Our framework is evaluated on a production HPC system and a testbed HPC cluster. We demonstrate that our proposed framework achieves a 0.95 F1-score using 28x fewer labeled samples compared to a supervised baseline using the whole active learning dataset, even when there are previously unseen applications and application inputs in the test dataset.

As a next step, we plan to cover a scenario where *ALBADross* is deployed on a production HPC system. The goal will be to design an interactive dashboard to make the querying process (i.e., asking for the label of the selected sample) easier for human annotators. To make this process more intuitive, we plan to incorporate some unsupervised techniques and domain heuristics together to point out the most important metrics. Another direction we plan to explore is to design a custom query strategy for multivariate time series data to further reduce the necessary labeled samples.

ACKNOWLEDGMENT

This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

[1] J. Gao *et al.*, "Sunway supercomputer architecture towards exascale computing: analysis and practice," *Science China Information Sciences*, vol. 64, no. 4, pp. 1–21, 2021.

[2] Y. Zhang, T. Groves, B. Cook, N. J. Wright, and A. K. Coskun, "Quantifying the impact of network congestion on application performance and network metrics," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 162–168.

[3] S. Chunduri *et al.*, "Run-to-run variability on xeon phi based cray xc systems," in *SC'17: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[4] V. J. Leung, M. A. Bender, D. P. Bunde, and C. A. Phillips, "Algorithmic support for commodity-based parallel computing systems." Sandia National Laboratories, Tech. Rep., 2003.

[5] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *IEEE/IFIP international conference on dependable systems and networks (DSN)*, 2013, pp. 1–12.

[6] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *SC'13: IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[7] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating i/o interference in hpc systems through cross-application coordination," in *IEEE 28th international parallel and distributed processing symposium*, 2014, pp. 155–164.

[8] M. Snir *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[9] J. Brandt *et al.*, "Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example," in *IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2010, pp. 2–7.

[10] A. Agelastos *et al.*, "Toward rapid understanding of production HPC applications and systems," in *IEEE International Conference on Cluster Computing*, 2015, pp. 464–473.

[11] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[12] R. Ahad, E. Chan, and A. Santos, "Toward autonomic cloud: Automatic anomaly detection and resolution," in *International Conference on Cloud and Autonomic Computing*, 2015, pp. 200–203.

[13] J. Brandt *et al.*, "Enabling advanced operational analysis through multi-subsystem data integration on trinity." Sandia National Laboratories, Tech. Rep., 2015.

[14] L. Zhang, X. Xie, K. Xie, Z. Wang, Y. Lu, and Y. Zhang, "An efficient log parsing algorithm based on heuristic rules," in *International Symposium on Advanced Parallel Processing Technologies*. Springer, 2019, pp. 123–134.

[15] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Mining console logs for large-scale system problem detection." *SysML*, vol. 8, pp. 4–4, 2008.

[16] B. L. Dalmazo, J. P. Vilela, P. Simoes, and M. Curado, "Expedite feature extraction for enhanced cloud anomaly detection," in *IEEE/IFIP Network Operations and Management Symposium*, 2016, pp. 1215–1220.

[17] S. Jin, Z. Zhang, K. Chakrabarty, and X. Gu, "Accurate anomaly detection using correlation-based time-series analysis in a core router system," in *IEEE International Test Conference (ITC)*, 2016, pp. 1–10.

[18] O. Tuncer *et al.*, "Diagnosing performance variations in HPC applications using machine learning," in *International Supercomputing Conference*. Springer, 2017, pp. 355–373.

[19] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 440–453.

[20] G. Wang, J. Yang, and R. Li, "An anomaly detection framework based on ica and bayesian classification for iaas platforms," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 10, no. 8, pp. 3865–3883, 2016.

[21] A. Borghesi, A. Bartolini, M. Lombardi *et al.*, "Anomaly detection using autoencoders in high performance computing systems," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, p. 9428–9433, 2019, arXiv: 1811.05269.

[22] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems," *Engineering Applications of Artificial Intelligence*, vol. 85, p. 634–644, 2019.

[23] B. Aksar, Y. Zhang, E. Ates, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Proctor: A semi-supervised performance anomaly diagnosis framework for production hpc systems," in *International Conference on High Performance Computing*. Springer, 2021, pp. 195–214.

[24] B. Settles, "Active learning literature survey," 2009.

[25] D. Angluin, "Queries and concept learning," *Machine learning*, vol. 2, no. 4, pp. 319–342, 1988.

[26] Y. Freund, H. S. Seung, E. Shamir, and N. Tishby, "Selective sampling using the query by committee algorithm," *Machine learning*, vol. 28, no. 2, pp. 133–168, 1997.

[27] Y. Wang *et al.*, "Practical and white-box anomaly detection through unsupervised and active learning," in *29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–9.

[28] S. Russo, M. Lürig, W. Hao, B. Matthews, and K. Villez, "Active learning for anomaly detection in environmental data," *Environmental Modelling & Software*, vol. 134, p. 104869, 2020.

[29] J. Chen, D. Zhou, Z. Guo, J. Lin, C. Lyu, and C. Lu, "An active learning method based on uncertainty and complexity for gearbox fault diagnosis," *IEEE Access*, vol. 7, pp. 9022–9031, 2019.

[30] Y. Lei, J. Lin, Z. He, and M. J. Zuo, "A review on empirical mode decomposition in fault diagnosis of rotating machinery," *Mechanical systems and signal processing*, vol. 35, no. 1-2, pp. 108–126, 2013.

[31] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, "Robust random cut forest based anomaly detection on streams," in *International conference on machine learning*. PMLR, 2016, pp. 2712–2721.

[32] T. Wu and J. Ortiz, "Rlad: Time series anomaly detection through reinforcement learning and active learning," *arXiv preprint arXiv:2104.00543*, 2021.

[33] C. Xie, W. Xu, and K. Mueller, "A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 215–224, 2018.

[34] T. Pimentel, M. Monteiro, A. Veloso, and N. Ziviani, "Deep active learning for anomaly detection," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[35] H. Bodor, T. V. Hoang, and Z. Zhang, "Little help makes a big difference: Leveraging active learning to improve unsupervised time series anomaly detection," *arXiv preprint arXiv:2201.10323*, 2022.

[36] C. Nixon, M. Sedky, and M. Hassan, "Salad: An exploration of split active learning based unsupervised network data stream anomaly detection using autoencoders," 2021.

[37] O. Tuncer *et al.*, "Online diagnosis of performance variation in hpc systems using machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 883–896, 2018.

[38] E. Ates *et al.*, "Taxonomist: Application detection through rich monitoring data," in *European Conference on Parallel Processing*. Springer, 2018, pp. 92–105.

[39] E. Baseman, S. Blanchard, N. DeBardeleben, A. Bonnie, and A. Morrow, "Interpretable anomaly detection for monitoring of high performance computing systems," in *Outlier Definition, Detection, and Description on Demand Workshop at ACM SIGKDD*, 2016.

[40] A. Ahmadzadeh, K. Sinha, B. Aydin, and R. A. Angryk, "Mvts-data toolkit: A python package for preprocessing multivariate time series data," *SoftwareX*, vol. 12, p. 100518, 2020.

[41] M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr, "Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package)," *Neurocomputing*, vol. 307, pp. 72–77, 2018.

[42] J. M. Yentes, N. Hunt, K. K. Schmid, J. P. Kaipust, D. McGrath, and N. Stergiou, "The appropriate use of approximate entropy and sample entropy with short data sets," *Annals of biomedical engineering*, vol. 41, no. 2, pp. 349–365, 2013.

[43] P. Welch, "The use of fast fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms," *IEEE Transactions on audio and electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.

[44] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[45] M. Christ, A. W. Kempa-Liehr, and M. Feindt, "Distributed and parallel time series feature extraction for industrial big data applications," *CoRR*, vol. abs/1610.07717, 2016. [Online]. Available: http://arxiv.org/abs/1610.07717

[46] D. H. Bailey *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *ACM/IEEE conference on Supercomputing*, 1991, pp. 158–165.

[47] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[48] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke-a massively parallel transport mini-app," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[49] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[50] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

[51] N. Petersson and B. Sjögreen, "Sw4 v1.1 [software]," Computational Infrastructure for Geodynamics, 2014.

[52] "Exascale proxy applications." [Online]. Available: https://proxyapps.exascaleproject.org/

[53] E. Ates *et al.*, "HPAS: An HPC performance anomaly suite for reproducing performance variations," in *ACM Proceedings of the 48th International Conference on Parallel Processing*, 2019, p. 1–10.

[54] G. C. Cawley, "Baseline methods for active learning," in *Active Learning and Experimental Design workshop In conjunction with AISTATS 2010*. JMLR Workshop and Conference Proceedings, 2011, pp. 47–57.

[55] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[56] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the ACM Int. Conf. on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[57] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[58] T. Danka and P. Horvath, "modal: A modular active learning framework for python," *arXiv preprint arXiv:1805.00979*, 2018.