Runtime Performance Anomaly Diagnosis in Production HPC Systems Using Active Learning

Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K. Coskun

Abstract—With the increasing scale and complexity of High-Performance Computing (HPC) systems, performance variations in applications caused by anomalies have become significant bottlenecks in system health and operational efficiency. As we move towards exascale systems, these variations become more prominent due to the increased sharing of resources. Such variations lead to lower energy efficiency and higher operational costs. To mitigate these problems, one must quickly and accurately diagnose the root cause of the anomalies at scale. One way to evaluate system health and identify the underlying causes is by manually examining certain performance metrics in telemetry data or using rule-based methods. Due to the daily size of telemetry data reaching terabytes and the fact that the numeric telemetry data contains thousands of metrics, manual analysis of telemetry to diagnose problems becomes challenging. Given these limitations, Machine Learning (ML)-based approaches have been gaining popularity as they have been shown to be effective and practical in diagnosing previously encountered performance anomalies. One primary challenge for supervised ML models is that they require a significant amount of labeled samples during training. However, obtaining many labels for anomalies is extremely difficult and costly, considering anomalies occur infrequently and real-world numeric system telemetry data is hard to label since it contains thousands of metrics. This paper proposes a novel active learning-based framework that diagnoses performance anomalies (i.e., identifying the type of an anomaly) in HPC systems at runtime using significantly fewer labeled samples compared to state-of-the-art ML-based approaches. We show that the proposed framework achieves the same F1-score compared to a supervised approach using much fewer labeled samples (i.e., 16x fewer samples for achieving a 0.78 F1-score, 11x fewer samples for achieving a 0.82 F1-score), even when there are previously unseen applications and application inputs in the test dataset.

Index Terms—anomaly diagnosis, active learning, machine learning, high-performance computing

1 Introduction

Modern large-scale computing systems are highly complex and parallel systems that perform a multitude of intricate operations simultaneously. They are vital for a variety of societal and scientific applications. Unfortunately, the increased usage of shared resources and the scale of systems often lead to substantial resource contention, resulting in performance variability and decreased efficiency. For example, it is possible to observe up to 8 times increase in job execution times [1] and more than a 70% variation in application performance even with the same input deck [2].

Performance variabilities can be caused by either software or hardware-related *anomalies*, such as network contention [3], OS jitter [4], firmware bugs [5], memory leakage [6], fluctuating CPU frequencies [7], or orphan processes that are leftover from previous jobs [8]. Detecting these anomalies is challenging because they may not directly result in failures, making them more elusive than outright failures. However, promptly identifying and diagnosing the root cause of performance anomalies is crucial for the energy and power efficiency of large-scale High-Performance Computing (HPC) systems. Today, HPC systems leverage various monitoring frameworks to understand how resources (i.e., memory, CPU, GPU, etc.) are utilized during application runs [9], [10], [11]. Anomalous behavior during the execution of an application is reflected in various forms of data, such as performance metrics, system logs, and traces. One method of detecting and diagnosing performance anomalies is manually analyzing these metrics, logs, and traces, which relies on human expertise. However, as HPC systems grow in size and complexity, this approach would necessitate investigating billions of data points daily, rendering it infeasible and impractical for humans. In addition, due to their complex nature, certain anomalies may be difficult to detect or differentiate, even by HPC experts, as they may manifest themselves across multiple performance metrics in a convoluted manner. A more efficient approach for anomaly diagnosis is automated performance analysis techniques, with Machine Learning (ML) being an auspicious method [12], [13], [14].

In this paper, we focus on ***diagnosing*** the type of performance anomaly (e.g., memory leakage or CPU contention) while an application is running instead of solely determining whether the run is anomalous or healthy. To accomplish this, we assume the availability of a few labeled *samples**, classified as either healthy or anomalous, with a specific type of anomaly. A variety of ML frameworks have been introduced, such as those utilizing Neural Networks [15]

- *Burak Aksar, Efe Sencan, Brian Kulis, Manuel Egele, and Ayse K. Coskun are with Electrical and Computer Engineering Department, Boston University.*
  E-mail: baksar, esencan, bkulis, megele, acoskun@bu.edu
- *Ben Schwaller, Omar Aaziz, Vitus J. Leung, and Jim Brandt are with Sandia National Laboratories. E-mail: bschwal, oaaziz, vjleung, brandt@sandia.gov*

*A sample is a vector (1 x N features) generated by extracting features of multivariate telemetry data (Timestamps x M metrics) collected from a compute node during an application run.

and Support Vector Machines (SVMs) [16], for detecting or diagnosing anomalies in a supervised setting. Although these supervised ML frameworks achieve high classification performance, their performance is heavily dependent on the availability of labeled data. However, the numeric telemetry data (such as CPU usage, memory consumption, network traffic, and disk I/O operations) collected from production HPC systems are usually unlabeled, and it is expensive for HPC administrators to provide labels for such a large amount of data. Several semi-supervised frameworks have been developed to mitigate this issue, and they achieve satisfactory detection or diagnosis scores using a modest size of labeled samples (e.g., [14], [17]). These semi-supervised frameworks either wait until the application run is completed (i.e., no runtime diagnosis) or only leverage the existing labeled samples (i.e., not covering dynamic scenarios where users or system admins can provide additional labels for new samples).

This paper builds on our recent active learning-based framework, *ALBADross* [18], which diagnoses performance anomalies in compute nodes after an application run is completed using significantly fewer labeled samples compared to state-of-the-art ML-based frameworks. *ALBADross* incorporates active learning to minimize the number of labeled samples required during training and employs a supervised classifier to determine the root cause of anomalies. We run controlled experiments to collect labeled numeric telemetry data from application runs and train an ML model using only a small subset of the collected data. Then, active learning determines which sample should be labeled among thousands of unlabeled samples to achieve a satisfactory F1-score for anomaly diagnosis during the training phase. In this paper, our specific contributions are as follows:

- Redesign of *ALBADross* to diagnose performance anomalies at ***runtime*** using a minimum number of labeled samples.
- Using data collected from real applications on a production HPC system, our framework achieves the same F1-score compared to a supervised approach using much fewer labeled samples (i.e., 16x fewer samples for achieving a 0.78 F1-score, 11x fewer samples for achieving a 0.82 F1-score).
- Quantification of our framework's *robustness* when the test dataset contains previously unseen applications and application inputs.
- Investigation of overheads (e.g., inference and feature extraction times) incurred in a production system deployment scenario.

The remainder of the paper starts with an overview of the related work. Then, Section 3 introduces our proposed runtime anomaly diagnosis framework. In Section 4, we explain our experimental methodology. In Section 5, we present our results and conclude in Section 6.

2 Related Work and Background

This section overviews recent anomaly detection and diagnosis research in large-scale computing systems, focusing first on active learning for detection and then on ML techniques for analyzing HPC telemetry data. The primary classification problem we address is detecting and diagnosing performance anomalies in multivariate time series telemetry data gathered from computing nodes. These anomalies may stem from various sources, such as load imbalances, network congestion, hardware malfunctions, and software inefficiencies. Additionally, our objective is to diagnose these performance anomalies at runtime rather than relying on post-application run analysis. The ability to rapidly detect and diagnose anomalies at runtime is crucial, considering that execution times for HPC applications can range from several days to weeks.

2.1 Active Learning and Anomaly Detection

Given the limited availability of labeled data in most domains, active learning has become increasingly prevalent for anomaly detection tasks, as it can significantly reduce the number of labels required during model training while still achieving satisfactory classification accuracy. Active learning is a semi-supervised learning paradigm based on iteratively querying the label of a data point provided by the oracle (e.g., a human annotator). The objective of active learning is to minimize the number of queries to the oracle (i.e., the person who provides the label) during the training by selecting the most informative data points from the unlabeled dataset and using them to train the ML model [19].

Active learning has been studied in various anomaly detection tasks over the years to improve classification performance when the number of labeled data is limited. For example, Huang et al. propose an online anomaly detection framework for cloud applications by combining uncertainty-based active learning query strategies with a variational autoencoder [20]. Wang et al. introduce ActiveMTSAD for detecting anomalies in key performance indicator data, where the data distribution changes over time due to continuous integration and deployment cycles [21]. Their framework consists of an unsupervised anomaly detector and active learner with three different feedback strategies (denominator penalty, negative penalty, and metric learning) and achieves over 0.95 F1-score in anomaly detection using 0.2% of the available labels. Another study by Li et al. presents a framework that combines active learning with variational autoencoders to detect anomalies in monitoring data collected from eBay's search services [22]. Their framework demonstrates an F1-score of up to 0.96 using only 3% of labeled data. Khowaja introduces an approach that incorporates quality learning characteristics into active learning, allowing the network to either predict or request the label of a data point during the training stage [23]. They leverage sparse autoencoders and long short-term memory with action-value functions to classify malware applications using a small number of labeled data points.

Despite active learning being a prevalent method in various anomaly detection tasks, there have been few studies that have applied it to the detection of performance anomalies in HPC systems. For instance, Xie et al. use call-stack trees to represent application executions as vector embeddings. Then, they combine active learning with a one-class SVM to detect anomalies [24]. While these active learning-based techniques yield promising results, our objective extends beyond simple detection. We aim to diagnose specific types of anomalies during the course of an application's execution.

2.2 ML-Based HPC Monitoring Analytics

As HPC systems generate terabytes of telemetry data daily, automated performance analytics become a vital component in system management. In particular, there has been a recent trend in utilizing ML-based approaches for detecting and diagnosing performance anomalies in HPC systems. For instance, Tuncer et al. introduce a supervised ML-based framework that first extracts statistical features from the collected telemetry data, then utilizes tree-based ML algorithms to diagnose the root causes of the anomalies [25]. Ates et al. use a random forest model to identify different applications running on supercomputers [26]. Klinkenberg et al. capture descriptive statistics from monitoring data and employ a supervised ML classifier to detect anomalous compute nodes [27]. While these methods achieve auspicious classification performance, they require a significant amount of labeled data during the training phase. However, in real-world systems, the collected telemetry data are mostly unlabeled.

To address this challenge, Borghesi et al. introduce a semi-supervised autoencoder-based framework that detects anomalies in compute nodes by learning healthy node characteristics [28]. However, their method cannot perform anomaly diagnosis (i.e., they do not identify the root causes of the anomalies). We propose a semi-supervised framework that first captures performance anomaly characteristics in an unsupervised manner, then integrates a supervised classifier to diagnose anomalies [14]. While this approach requires fewer labels than a supervised training setup, its performance is limited by the available labels.

In our recent work [18], we design an anomaly diagnosis framework *ALBADross* that combines an active learning-based query strategy and a supervised classifier to minimize the number of labeled samples required for achieving a target performance score. However, this framework necessitates that an application run is completed before any diagnosis can be made. Some HPC applications run for days or even weeks; therefore, diagnosing the anomalies at runtime is necessary to design more effective mitigation and resource management policies. In this paper, we redesign *ALBADross* for **runtime** anomaly diagnosis, and we investigate the impact of various deployment-related parameters (e.g., feature extraction, model training, and inference) for a production system deployment scenario.

3 *ALBADross* 2.0: Runtime Diagnosis

Our primary objective is to **diagnose** the underlying causes of performance anomalies at **runtime** in an application-agnostic way as much as possible. We mainly focus on identifying and understanding anomalies that do not result in crashes or errors, as they can be more elusive and challenging than failures. To achieve this goal, in this paper, we redesign our active learning-based framework, *ALBADross* [18].

Figure 1 shows the redesigned version of *ALBADross*. Compared to the initial version of *ALBADross* [18], we make several modifications to enhance its functionality. The redesigned version introduces new active learning query strategies, as well as the capability for runtime anomaly diagnosis. First, we collect multivariate numeric telemetry data from compute nodes while running applications with and without synthetically introduced anomalies. We then divide telemetry data into equal-length windows and extract features of each window. Then, the initial model is trained with the available windows in the labeled dataset. Next, the active learning module examines windows in the unlabeled dataset and selects a subset of windows for annotation based on various query strategies. Finally, the model is retrained using the newly labeled windows. Our system is independent of the monitoring framework being employed. The following sections cover these stages more deeply.

3.1 Window Generation

To diagnose performance anomalies at runtime, our framework divides the telemetry data into equal-length windows of a specified size, denoted as w. Each window is a 2D vector: [w x M metrics]. S is a hyperparameter controlling skip interval, and its value is determined offline based on the trade-off between computational time and delay in anomaly diagnosis. As the window size increases, we have fewer windows, hence, lower CPU and memory requirements, but the diagnosis delay is higher. Therefore, the value of w should be carefully chosen depending on the requirements.

3.2 Feature Extraction

We apply feature extraction for each window to gain insight into the underlying temporal and spectral characteristics of time series. We use two different open-source feature extractors: *TSFRESH* [29] and *MVTS* [30]. *TSFRESH* computes 794 features for each metric in raw telemetry data. For instance, C3 statistics [31], which measures the non-linearity of the time series, Benford correlation [32], which detects anomalous patterns, and descriptive statistics such as mean, standard deviation, and maximum. *MVTS* computes 48 statistical features for each metric. Some extracted features include descriptive statistics, the absolute difference between descriptive statistics, and their derivatives.

3.3 Feature Selection

To save the computational time spent during the model training phase and achieve better anomaly diagnosis performance, we use the Chi-Square approach [33] to select a subset of metrics after the feature extraction stage. Chi-Square is a statistical test to measure two events' independence. Given the observed values and the expected value, Chi-Square computes how the observed value deviates from the expected value. We choose features that strongly depend on our class labels, which we determine by selecting the ones with the highest Chi-Square values.

3.4 Hyperparameter Search and Initial Training

At this stage, the goal is to determine the best hyperparameters for supervised models prior to leveraging active learning. The initially labeled dataset assumes a single labeled sample representing each distinct combination of application-anomaly pair is available. Using this dataset, we perform a hyperparameter search for each model through a grid search over several cross-validation (CV) sets.

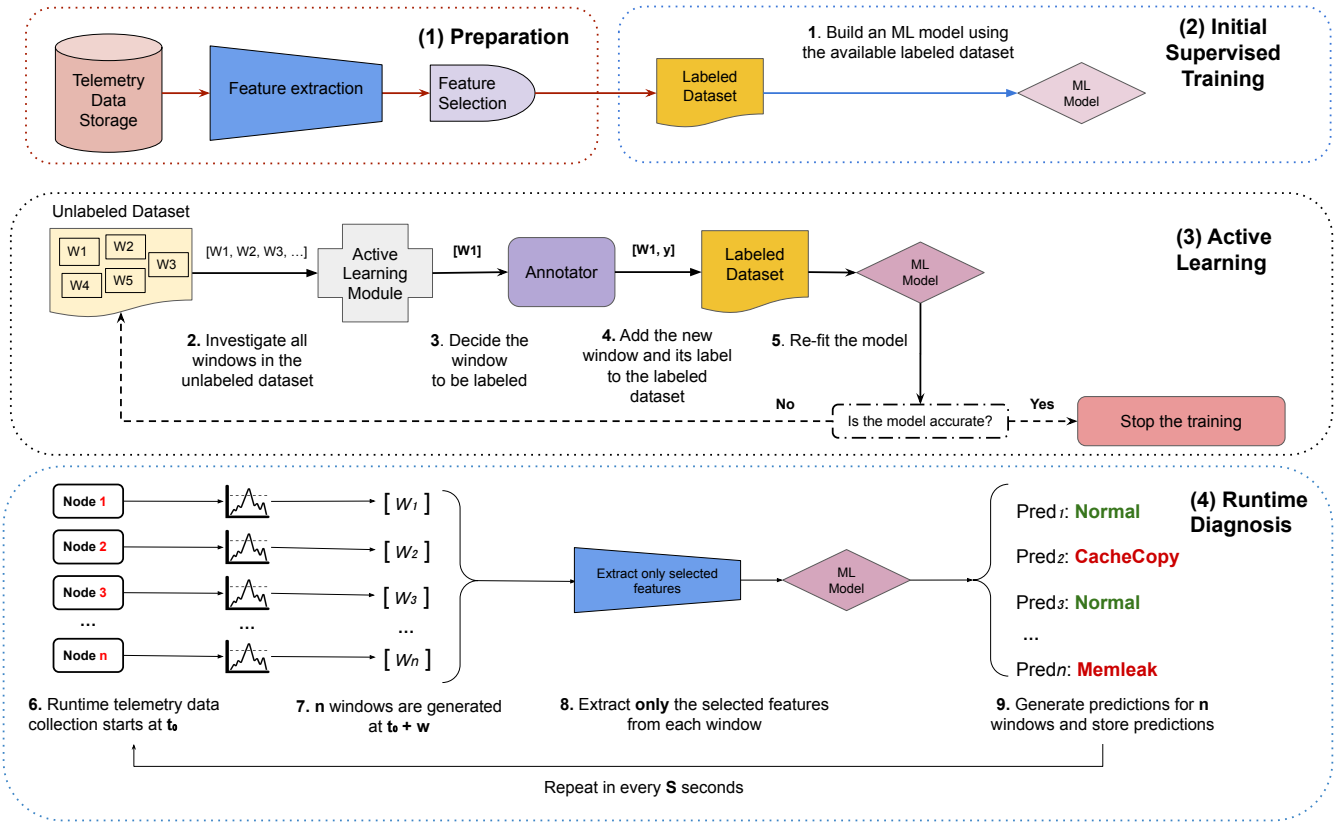


Fig. 1. The redesigned architecture of *ALBADross*. First, we divide the telemetry data into equal-length windows (W) based on the window size (w) and skip interval (S). Then, we extract features and select useful features based on statistical measures. We then train a supervised model using the available labeled samples. The active learning module determines which windows should be labeled from the unlabeled dataset, and then an annotator provides labels. We continue this process until we either meet the desired performance score or spend the maximum number of samples our budget allows for labeling. At runtime, we continuously monitor every compute node. Assuming the model is deployed at t_0 and the system has n compute nodes, n windows are generated in total (denoted as $W_1, W_2, ..., W_n$) at $t_0 + w$. We only extract the selected features during the preparation stage and provide the label (e.g., *memleak*, *normal*) for each window. Steps 8 and 9 are repeated whenever a new batch of windows arrives every S seconds.

3.5 Active Learning

After completing the initial training stage, we use an active learning query strategy to determine the most informative windows in the unlabeled pool. The informativeness of a window is determined by the selected active learning query strategy. We query 50 windows simultaneously at each query iteration and then learn their labels from the annotator. This process is referred to as batch mode active learning. It is useful when the dataset is large and frequently retraining the ML model is computationally costly. In the labeling process, we hold two key assumptions. First, we assume that annotators provide only correct labels for the selected windows or samples. Second, we assume that annotators possess either a systematic method or domain expertise for labeling the selected windows or samples. We plan to examine these issues in future work, as noted in Section 6. We experiment with two different sampling methods: uncertainty sampling and query-by-committee (QBC) [34], which we describe in the following sections.

3.5.1 Uncertainty Sampling

In uncertainty sampling, the active learner chooses a sample to be labeled based on the model's most uncertain predictions. We employ all available query strategies for classification tasks in the selected software package. We represent class probabilities with $\hat{s_i} = [p_1, p_2, p_3, ..., p_k]$, where i denotes the sample's index, p_k is the probability for the k-th class, and $\hat{s_i}$ contains all class probabilities for the sample. Assume we have the following class probabilities for three different samples:

$$\hat{s_1} = [0.1, 0.85, 0.05];\ \hat{s_2} = [0.6, 0.3, 0.1];\ \hat{s_3} = [0.39, 0.61, 0.0]$$

Classification Uncertainty computes the uncertainty probabilities of samples as follows:

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

where x is the instance to be predicted, and y is the most likely class prediction. It then selects the sample with the highest uncertainty probability. The uncertainty probabilities of the above example are $U_{list} = [0.15, 0.4, 0.39]$, and the selected example is the second one.

Classification Margin computes the class probabilities of each sample, and it calculates the difference between the first and second highest class probabilities as follows:

$$M(x) = P(y_1|x) - P(y_2|x), \tag{2}$$

where y_1 and y_2 are the first and second most likely classes. It then selects the sample with the lowest margin.

Classification Entropy computes the entropy of each sample based on the prediction probabilities of each sample:

$$H(x) = -\sum_{y} P(y|x)\log(P(y|x)). \quad (3)$$

Then, it selects the sample with the highest entropy.

3.5.2 Query-by-Committee

QBC determines the informativeness of an unlabeled sample based on the disagreement level of multiple classifiers. This can alleviate the drawbacks of having a single classifier biased towards certain classes while measuring uncertainty. There are several strategies to determine the disagreement level. We employ all available query strategies for classification tasks in the selected software package.

Consensus Entropy Sampling (CES) computes the class probabilities of unlabeled samples for each supervised classifier. It then takes the average of these class probabilities, called consensus probabilities. Finally, it computes the entropy of the consensus probabilities and picks the sample with the highest entropy.

Vote Entropy Sampling (VES) computes the class probabilities for each sample. Then, it computes the probability ratios of predicted labels for each sample. Then, it computes the entropy of these probability distributions and selects the sample with the highest entropy.

Maximum Disagreement Sampling (MDS) computes the vote probabilities for each classifier and then obtains the consensus probabilities. Then, it computes the Kullback-Leibler divergence [35] of each classifier with respect to the consensus prediction. Finally, it selects the sample with the largest value.

3.6 Runtime Anomaly Diagnosis

Since some HPC applications can run for extended periods, it is essential to diagnose anomalies at runtime. While training the model, we employ an active learning query strategy and monitor the training process until a certain condition is met, such as reaching the maximum allowed queries or target diagnosis score. Once this condition is achieved, the training is finalized, and the trained model is stored. At runtime, we continuously collect telemetry data from each compute node. Then, we create equal-length windows and extract only the selected features. The trained model outputs the anomaly type for each window if the window is anomalous; otherwise, it classifies as normal.

4 Experimental Methodology

The first subsection discusses the applications that run on the target production system. The following subsections explain the monitoring framework and synthetic anomalies we inject to mimic common performance variations in HPC systems. We conclude the section by providing the implementation details.

TABLE 1
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

4.1 Production System and Selected Applications

We run our applications on a production HPC system called *Eclipse* located at Sandia National Laboratories. Eclipse has 1,488 compute nodes and a peak performance of 1.8 petaflops. Each node has 128GB of RAM with two sockets, and each socket has 18 E5-2695 v4 CPU cores. We run six applications: *LAMMPS*, *HACC*, *sw4*, *ExaMiniMD*, *SWFFT*, and *sw4lite*. Amongst these, three of them are real applications: *LAMMPS*, a molecular dynamics simulation focusing on materials modeling [36]; *HACC*, an extreme-scale cosmological simulation [37]; and *sw4*, a popular 3D seismic model [38]. The remaining three, *ExaMiniMD*, *SWFFT*, and *sw4lite*, are proxy applications from the ECP Proxy Apps Suite [39]. We list all applications used in our experiments in Table 1. We run each application for 20-45 minutes on 4, 8, and 16 nodes, where each application has a different input deck for each unique node count, i.e., input one is designed for running the application on four nodes, whereas input two is designed to run the application on eight nodes.

TABLE 2
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

4.2 Monitoring Framework

We use the Lightweight Distributed Metric Service (LDMS) [10] to collect telemetry data from applications at runtime. LDMS is a monitoring framework capable of gathering, transferring, and storing telemetry data on large-scale distributed systems with low overhead. It collects data from different subsystems and performance counters at the second granularity for a specific compute node. Note that our framework is not limited to LDMS and can be customized to work with different monitoring frameworks. Some example metrics from different subsystems are memory (e.g., free, active, inactive memory), CPU (e.g., user and idle time, I/O wait time), network (e.g., received/transmitted packets, average packet size, link status), shared file system (e.g., open, read, write counts), virtual memory (e.g., free, active, and inactive pages). We gather 806 metrics at a rate of 1Hz from each compute node. However, we do not use *per-core* related metrics (e.g., per_core_cpu_enabled8, per_core_guest8, etc.) as we observe significant fluctuations in them for the same

application run with the same input. We use node-level CPU features as they exhibit greater consistency. After dropping per-core metrics, we have 156 metrics.

4.3 Synthetic Performance Anomalies

To assess the performance of our framework, a dataset with ground truth labels is necessary. However, obtaining such a dataset from production HPC systems is challenging due to the infrequency of anomalies and complexity of the data. Several fault or anomaly injection tools (e.g., FINJ [40], GPCNet [41], and PFault [42]) are available. GPCNet employs a range of configurable patterns to induce congestion across the network systematically. Meanwhile, PFault mimics failure conditions for individual storage nodes within the parallel file system by leveraging a set of predetermined fault models. While existing tools typically focus on a single subsystem or solely inject faults that result in program termination, we aim to identify performance anomalies that target multiple subsystems (e.g., memory, CPU), resulting in diminished performance.

We use the HPC Performance Anomaly Suite (HPAS) [43], an open-source performance anomaly suite reproducing common performance anomalies in production HPC systems. In HPAS, synthetic anomalies aim to impact five main subsystems: CPU, cache, memory, network, and shared storage. These anomalies operate through processes that run in user space, eliminating the need for any hardware or kernel modifications. The details of injected anomaly types and their anomalous behavior are provided in Table 2. We also carry out experiments that include anomalies related to I/O and network. Due to significant contention caused by I/O-related anomalies, system administrators terminate the runs. The network anomaly only causes contention when applications operate on two compute nodes; therefore, it is not included in our experiments. While executing an application that runs on multiple compute nodes, we run a synthetic anomaly on every node that the application uses. If an anomaly is injected for that application run, the telemetry data for each compute node is labeled with an anomaly type; otherwise, it is labeled as *healthy*. We also run the same anomaly with two or three intensity settings.

4.4 ML Models and Baselines

We choose the ML models used in the HPC domain to evaluate the diagnosis performance: SVM, Random Forest (RF), Multi-Layer Perceptron (MLP), and Light Gradient Boosting Machine (LGBM). To test the performance of query strategies, we implement the random selection, *Random*, as a baseline [44]. There are other active learning-based techniques we consider for baselines. Huang et al. [20] and Wang et al. [21] employ unsupervised anomaly detection methods combined with active learning to classify time series windows as either healthy or anomalous. Our objective, however, is to identify types of anomalies within these windows. Furthermore, our training data originates from multiple compute nodes, unlike these techniques that depend on a singular data source with a continuous timestamp sequence. Given these differences, we opt not to benchmark our method against theirs.

4.5 Implementation Details

This section details the parameters and design decisions made during the implementation phase.

4.5.1 Telemetry Data Collection

We gather telemetry data while running applications in the form of multivariate time series data R^{TxM} where T is the number of timestamps that belong to that application run, and M is the number of metrics. After the data collection, we remove the first and last 60 seconds of telemetry data of each application run, as some metrics may deviate significantly from expected values during the initialization and termination phases. The choice of 60 seconds is based on investigating raw telemetry data and is specific to the applications

TABLE 3
The performance summary of the supervised classifiers in terms of F1-score, FAR, and AMR with TSFRESH and MVTS feature extractors. For each classifier, we report the model's performance when trained in *ALTD* and *5-fold CV* settings. We report the performance on the same test dataset for each training setup. The bold text shows the best-performing models.

			F1-score (Macro Avg)		False Alarm Rate (%)		Anomaly Miss Rate (%)	
			mean	max	mean	max	mean	max
Feature Extractor	Model	Configuration						
MVTS	**LGBM**	5-Fold CV	90.41	91.12	0.00	0.00	0.05	0.20
		ALTD	82.21	82.21	0.00	0.00	0.28	0.28
	MLP	5-Fold CV	87.36	88.07	0.41	0.54	0.16	0.41
		ALTD	71.95	71.95	0.02	0.02	3.15	3.15
	RF	5-Fold CV	83.23	83.74	0.02	0.02	0.34	0.51
		ALTD	71.74	71.74	0.00	0.00	6.35	6.35
	SVM	5-Fold CV	69.88	74.23	2.37	4.75	2.48	3.63
		ALTD	51.24	51.24	0.05	0.05	23.93	23.93
TSFRESH	LGBM	5-Fold CV	88.93	89.41	0.22	0.33	0.15	0.41
		ALTD	82.45	82.45	0.00	0.00	0.22	0.22
	MLP	5-Fold CV	83.31	84.23	0.04	0.09	0.20	0.42
		ALTD	73.80	73.80	0.00	0.00	0.31	0.31
	RF	5-Fold CV	84.00	84.58	0.00	0.00	0.24	0.47
		ALTD	81.58	81.58	0.00	0.00	1.19	1.19
	SVM	5-Fold CV	63.04	69.02	8.55	13.54	7.91	12.59
		ALTD	50.85	50.85	0.01	0.01	23.61	23.61

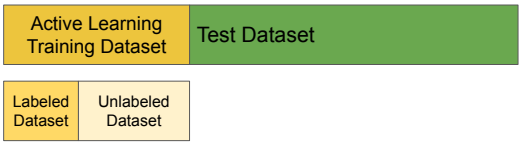


Fig. 2. Splitting the dataset into training and test datasets. The labeled dataset represents the initial state for the supervised training stage, where one compute node telemetry data from each application-anomaly pair exists. The unlabeled dataset is used to determine the samples to be labeled.

used. Some LDMS metrics accumulate the raw values, and since we are interested in the relative change, we calculate the difference between each time step. As a final stage, we apply linear interpolation in every time series to fill in missing values, as some values in LDMS metrics may be lost during the collection stage due to sampling problems. After the preprocessing stage, we create equal-length windows of 60 seconds with a 15-second skip interval, such as [0 - 60], [15 - 75], and [30 - 90]. We train the same supervised classifier using both 45-second and 60-second windows and select the 60-second window because it yields a superior F1-score, a lower anomaly miss rate, and a reduced false alarm rate during evaluation [45]. After the window generation stage, we obtain 507,738 and 1,677,999 windows for our train and test datasets from 24,566 compute nodes, respectively.

4.5.2 Feature Extraction and Selection

To extract essential characteristics of raw time series data and obtain the most relevant features corresponding to the class labels, we apply feature extraction and feature selection, respectively. We use the *efficient* setting in the TSFRESH feature extractor, which generates 121,836 features. For the MVTS package, the total number of features we obtain after the feature extraction step is 6,396. Some example features that are generated per metric are: *absolute_energy*, *benford_correlation*, *fft_coefficient*, *kurtosis*, *maximum*, *minimum*, *quantile*, and *skewness*, etc. Unfortunately, it is not feasible to extract features from all available windows simultaneously due to memory constraints. To mitigate this problem, we process windows in smaller groups by submitting parallel jobs and then store the feature-extracted versions. To reduce the dimensionality of the feature-extracted time series data, we utilize the Chi-Square feature selection technique and select the top 2,000 features. The choice of 2,000 is based on the experimentation in our previous work [18].

4.5.3 Dataset Split and Hyperparameter Tuning

In Figure 2, we show how we split our dataset into training and test datasets. To mimic the production system scenario, we further divide the active learning training dataset into labeled and unlabeled datasets where the size of the unlabeled dataset is significantly larger than the labeled dataset. Note that the figure does not represent the actual sizes of the datasets. We use the labeled portion of the active learning training dataset to train classifiers. Initially, the labeled dataset consists of 30 compute node telemetry data since the Eclipse dataset has six applications and five labels. We use the unlabeled portion of the active learning training dataset to query the label of the most informative windows iteratively. After determining the windows to be queried, we remove it from the unlabeled dataset and add it to the labeled dataset. Then, we retrain our supervised ML model and test the model's performance on the test dataset.

We also maintain a 10% anomaly ratio (i.e., the number of anomalous samples divided by all samples) in the active learning training dataset. The choice of a 10% anomaly ratio is based on our observations on Eclipse. Before running synthetic anomalies, we first investigate the percentage of the application runs that show outlier characteristics (i.e., runs with execution time 1.5 interquartile range below the 25th percentile or 1.5 interquartile range above the 75th percentile in terms of execution time). We observe that the outlier ratio ranges between 2-7%. Therefore, we cap the anomaly ratio as 10% in our training dataset.

We experiment with different hyperparameters for each model to select the best-performing supervised ML classifier. We tune the hyperparameters by performing a 5-fold CV on the active learning training dataset. Since hyperparameter tuning takes too much time on the windowed dataset, we use a feature-extracted version of the raw time series while searching for the optimal parameters analogous to our prior work [18].

5 Evaluation

Our framework is evaluated across three distinct experimental scenarios to assess its performance, utilizing three key performance metrics: F1-score, false alarm rate (FAR), and anomaly miss rate (AMR). Despite each active learning query strategy utilizing unique informativeness metrics (detailed in Section 3.5) to select samples for labeling, our overarching goal remains uniform in focusing on these three metrics. The F1-score combines precision (how often the model is correct when it predicts a positive class) and recall (how often the model correctly identifies actual positive class instances). FAR is the percentage of healthy windows incorrectly classified as anomalies, i.e., the number of false positives divided by the number of false positives plus the number of true negatives. AMR is the percentage of anomalous instances incorrectly classified as healthy, i.e., the number of false negatives divided by the number of false negatives plus the number of true positives.

5.1 Selecting Supervised Classifiers

This experiment aims to determine the best-performing supervised models before applying active learning query strategies. We experiment with four different ML classifiers and report their F1-score, FAR, and AMR for two settings. In the *active learning training data* (*ALTD*) setting, we train the classifier with the samples in the active learning training dataset (Figure 2) and report the anomaly diagnosis performance in the test dataset. In the *cross validation* setting (*5-fold CV*), we apply 5-fold CV using the whole dataset, i.e., active learning training and test datasets.

Table 3 shows the F1-score, FAR, and AMR when the classifiers are evaluated in *5-fold CV* and *ALTD* settings. LGBM and RF are the top two performing models in terms of F1-score with *MVTS* and *TSFRESH* feature extractors, respectively. The anomaly diagnosis performance of SVM

TABLE 4
Evaluation of various ML models and query strategies with MVTS and TSFRESH feature extractors. *Random** is the baseline query strategy. For each setting, we report the F1-score, AMR, and FAR when the model learns the labels of 500, 1,000, and 2,000 windows from the unlabeled pool. The bolded models are the best-performing ones.

			F1-score (Macro Avg)				False Alarm Rate (%)				Anomaly Miss Rate (%)			
		Num. Windows	0	500	1000	2000	0	500	1000	2000	0	500	1000	2000
Feature Extractor	Model	Query Type												
MVTS	LGBM	Entropy	66.01	79.29	81.24	81.67	3.22	0.04	0.01	0.00	0.62	0.34	0.34	0.23
		Margin	66.07	81.15	81.95	81.27	3.15	0.01	0.00	0.00	0.64	0.70	0.26	0.23
		Random*	66.03	71.64	73.38	76.41	3.23	0.13	0.07	0.03	0.65	0.39	0.42	0.47
		Uncertainty	66.11	78.71	82.05	81.49	3.02	0.08	0.00	0.00	0.65	0.31	0.27	0.23
	RF	Entropy	56.02	62.42	65.71	68.61	19.56	0.40	0.34	0.09	2.55	1.87	0.93	0.67
		Margin	56.15	68.27	71.76	75.46	20.16	0.01	0.01	0.00	2.07	0.82	0.45	0.34
		Random*	56.09	56.34	56.81	57.86	20.87	0.27	0.12	0.03	2.18	6.43	6.17	6.30
		Uncertainty	56.08	64.36	67.68	73.21	17.84	0.44	0.27	0.18	2.52	0.89	0.55	0.45
TSFRESH	LGBM	Entropy	73.17	76.59	78.68	82.29	5.14	0.19	0.13	0.07	0.32	0.34	0.32	0.29
		Margin	73.16	79.93	82.23	82.32	5.26	0.03	0.01	0.00	0.31	0.37	0.33	0.27
		Random*	73.20	74.91	75.77	76.96	5.20	0.30	0.20	0.06	0.31	0.34	0.35	0.37
		Uncertainty	73.20	76.81	80.56	82.42	5.02	0.05	0.01	0.00	0.31	0.35	0.34	0.26
	RF	Entropy	74.49	74.88	76.16	78.57	0.06	0.00	0.00	0.00	0.29	0.34	0.30	0.29
		Margin	74.41	78.92	81.23	83.02	0.07	0.00	0.00	0.00	0.28	0.30	0.30	0.29
		Random*	74.13	75.09	75.37	75.60	0.09	0.02	0.01	0.00	0.28	0.34	0.38	0.40
		Uncertainty	74.23	76.40	77.85	78.32	0.06	0.00	0.00	0.00	0.28	0.29	0.29	0.29

is poor since we only experiment with the linear kernel due to a quadratic increase in the execution time with respect to the number of samples. Both LGBM and RF achieve the perfect FAR in *ALTD* setting with *MVTS* and *TSFRESH* feature extractors. When we use *MVTS* feature extractor and Chi-Square feature selection with 2000 features, LGBM and MLP are the best-performing models in the *5-fold CV* setting in terms of F1-score. RF has the same F1-score as MLP in *ALTD* setting, but its F1-score is slightly lower for the *5-fold CV* setting. In terms of FAR, LGBM and RF are the best-performing models. Even though RF's performance is slightly lower in *5-fold CV* setting, it is more important not to raise false alarms in the context of anomaly diagnosis. So, we choose LGBM and RF as the top-performing models.

5.2 Anomaly Diagnosis with Active Learning

This scenario aims to determine the minimum number of windows required to achieve a certain F1-score. We evaluate multiple active learning query strategies from two sampling methods: uncertainty sampling and QBC. We also compare their performances with the *Random* baseline and fully supervised settings. Active learning query strategies and the *Random* baseline starts with one sample per application-anomaly pair (i.e., approximately 30 samples). We then query 2000 windows for each method and report F1-score, AMR, and FAR on the same test dataset after each query. In each figure, the black dashed line shows the F1-score of a supervised classifier when the model is trained in *ALTD* setting, referred to as *F1-ALTD*. The red dashed line shows the minimum number of windows needed to reach F1-ALTD, referred to as *Min-Query-ALTD*. The purple dashed line shows the F1-score of a supervised classifier when the model is trained in a 5-fold CV setting, referred to as *F1-CV*.

Table 4 shows the anomaly diagnosis results of active learning with different query strategies. The margin query is the best strategy for both classifiers since it achieves

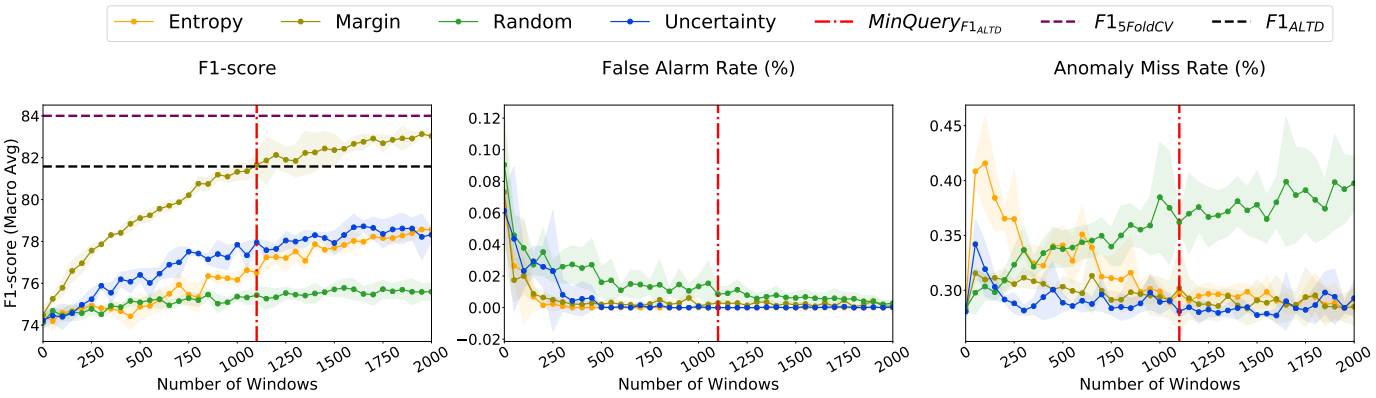


Fig. 3. The F1-scores, FARs, and AMRs of different uncertainty sampling query strategies, and the *Random* baseline for the first 2000 queried windows when using RF as a supervised classifier and TSFRESH as a feature extractor. Among uncertainty sampling strategies, margin sampling is the best-performing query strategy, reaching an F1-score of 82% with an additional 1100 windows.

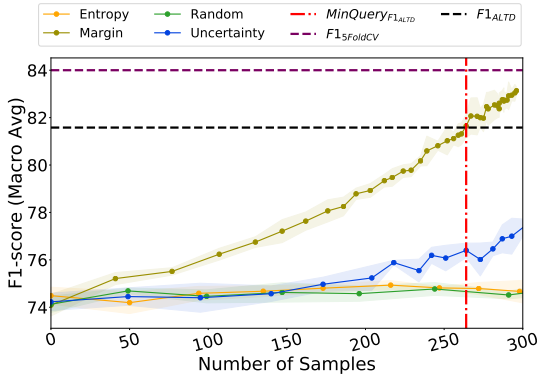


Fig. 4. The change in F1-score in terms of *samples* for different query strategies used in Figure 3. The margin query strategy reaches the F1-score of the supervised classifier trained in *ALTD* setting (5619 samples) by only querying the label of an additional 260 samples.

the highest F1-score with the least number of windows. LGBM with margin query strategy reaches an 81.15% F1-score using *MVTS*, and it reaches a 79.93% F1-score with *TSFRESH* using additional 500 windows, which is nearly the same F1-score (82.45%) when the model is trained in *ALTD* setting, which includes approximately 500,000 windows. To make a fair comparison, we gradually increase the number of windows and find that the model achieves the same performance with approximately 170,000 instead of 500,000. On the other hand, *Random* needs to query more than 10000 windows to achieve the same accuracy. This shows that active learning query strategies can be beneficial in determining which windows should be labeled. Figure 3 shows the change in the F1-score, FAR, and AMR with respect to the increasing number of windows when we use different query strategies and the *Random* baseline with *TSFRESH*. All active learning query strategies, especially margin, significantly outperform *Random* baseline by achieving a higher F1-score with fewer windows. Considering FAR and AMR, RF with margin query strategy achieves near-perfect results when using *TSFRESH*.

Even though it is possible to provide a label for a single window, it can be difficult for a human annotator to determine the health status of an application based on a single window, as this limited observation may not provide sufficient information. To ensure more reliable feedback, the annotator may need to examine larger periods of the application or even the entire run. We define the *sample* as the feature extracted version of the telemetry data collected from one compute node when an application runs. Our assumption is that all windows in a sample share the same label, based on our initial data collection strategy. As outlined in Section 4.5.2, each sample contains 2000 features, though not all are necessary for labeling. For example, an annotator may focus on specific memory usage metrics, such as free memory, utilizing their domain expertise to detect a memory leak and consequently label all windows in the sample as exhibiting a *memleak* anomaly. It is important to note that this paper does not directly address the labeling of high-dimensional telemetry data, a topic we reserve for future exploration. Initially, the margin query strategy reaches *F1-ALTD* using 1100 windows (Figure 3). However, to label 1100 windows, a human annotator should only provide the labels of 260 samples, as shown in Figure 4. This means that the actual cost of obtaining labels is significantly lower than the total number of queried windows if the annotator chooses to investigate samples.

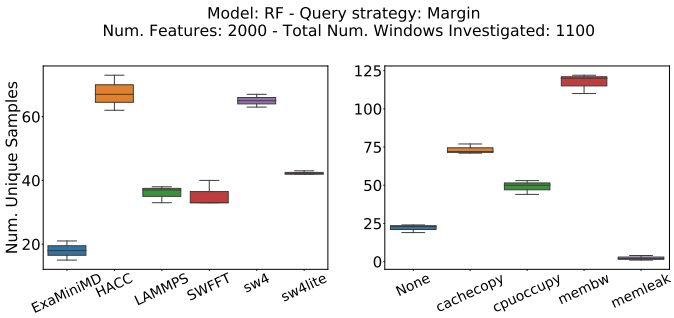


Fig. 6. The distribution of application and anomaly types within the first 1100 queried windows when using RF with margin sampling as a query strategy. The top two queried anomaly types are *cachecopy* and *membw*. The top two queried applications are *HACC* and *SW4*.

We also perform a drill-down analysis to understand which application and anomaly types are queried by the active learner. Figure 6 shows how many samples are selected from each application and anomaly type for RF with *TSFRESH* feature extractor. We observe that *cachecopy* and *membw* are the most frequently queried anomaly types. This shows that these anomalies created more confusion in the

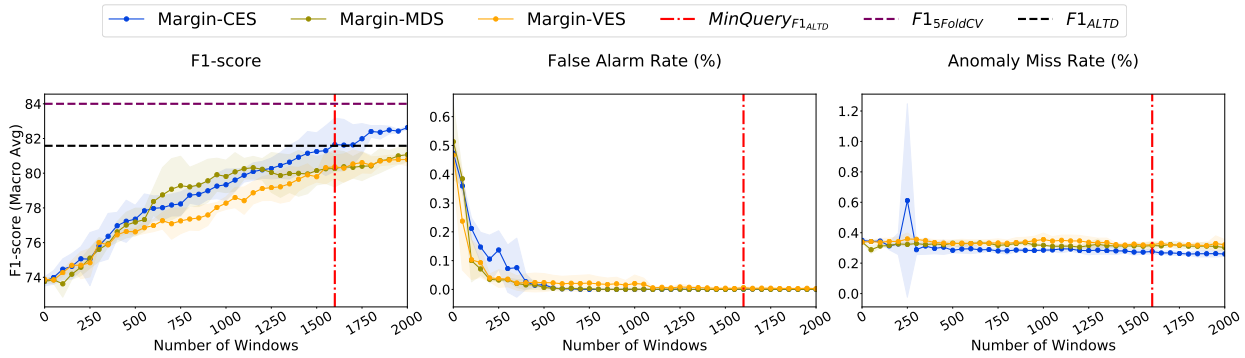


Fig. 5. The F1-scores, FARs, and AMRs of different QBC sampling query strategies, and the *Random* baseline for the first 2000 windows when using *TSFRESH* as a feature extractor. We use RF and LGBM as supervised classifiers for the two active learners and margin sampling as an uncertainty sampling query strategy. Margin-CES strategy is the best-performing strategy since it reaches an F1-score of 82% with fewer windows.

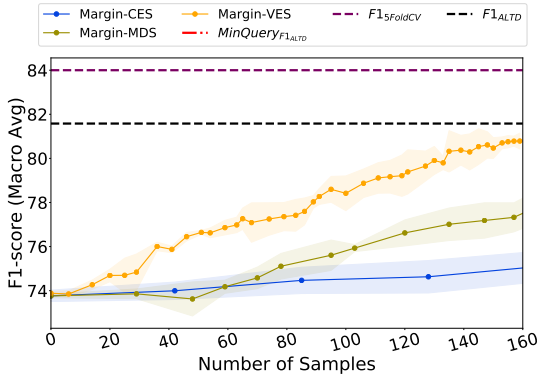


Fig. 7. The change in F1-score in terms of *samples* for different uncertainty sampling query strategies (Figure 5). Margin-VES sampling reaches the F1-score of a supervised classifier trained in *ALTD* setting by querying the label of an additional 159 samples.

model than other anomaly types; therefore, their label is queried more often. Among the HPC applications, *HACC* and *SW4* are the most queried application types. We also conduct a similar drill-down analysis for LGBM in the same setting. One key difference is that LGBM queries 3x more samples with *None* label, but it requires fewer samples for *membw* anomaly. Regarding application types, *HACC* and *SW4* are the most queried. It is expected to observe that each classifier prioritizes different characteristics.

We conduct a similar experiment for the QBC sampling query strategies. The main motivation is to leverage the strengths of two different classifiers to reduce further the number of samples that should be labeled. Based on previous drill-down analysis results, LGBM and RF prioritize different anomaly types. We initialize two active learners with LGBM and RF, respectively. Figure 5 shows the performance of QBC sampling query strategies in terms of F1-score, FAR, and AMR. In terms of achieving *F1-ALTD*, the margin-CES query strategy is the best one. However, margin-MDS and margin-VES query strategies also achieve a very close score. All strategies achieve a perfect FAR and 0.4% AMR. In Figure 7, we investigate the number of samples required to reach *F1-ALTD*. The margin-VES query strategy leverages entropy scores to determine which samples to label, which achieves the highest F1-score by querying the label of an additional 159 samples.

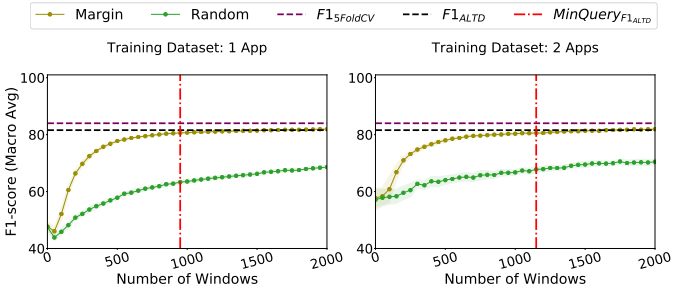


Fig. 8. The F1-scores, FAR, and AMR when the training dataset includes 1 and 2 known applications, while the test set has the remaining 4 applications. We repeat this experiment for all combinations. We reach *F1-ALTD* by querying the label of an additional 950 windows.

5.3 Robustness for Anomaly Diagnosis

In production HPC systems, collecting labeled samples for each combination of applications and application inputs is not feasible, as there may be thousands of combinations. Our previous work demonstrates a 30% drop in average F1-score and a 35x increase in FAR of a fully-supervised model when there are unseen applications in the test dataset [18]. Thus, we conduct experiments with unseen applications and inputs to gauge the robustness of our framework.

5.3.1 Previously Unseen Applications

To replicate the situation where there are previously unseen applications in the test dataset, we select four applications to be included in the test dataset and begin the initial training phase with the remaining two applications. However, it is worth noting that samples of all application types are present in the unlabeled pool, which means that if active learning determines that a particular sample is informative, it may still query the label of an application type that is present in the test dataset.

Figure 8 illustrates the change in F1-score in relation to the number of queried windows. On the left side of the plot, we initiated the initial training phase by having one labeled sample for all anomaly types of a single application (i.e., five labeled samples in total). The starting F1-score, in this case, was 49%, which is significantly lower compared to a scenario where we have one labeled sample for each application and anomaly pair. However, as we queried the label of 950 additional windows (505 samples), our framework achieves *F1-ALTD*. On the right side of the plot, we begin with one labeled sample for all anomaly types of two applications (i.e., ten labeled samples). The starting F1-score, in this case, was 58%, which is again lower when compared to a situation where we have labeled samples of all application and anomaly types. We achieve *F1-ALTD* by querying the label of an additional 1100 windows (508 samples), demonstrating that our framework is robust against previously unseen applications.

5.3.2 Previously Unseen Application Inputs

We conduct a similar experiment to the previously unseen application experiment, where the test dataset includes previously unseen application inputs. We then evaluate the robustness of our framework by measuring the F1-score with respect to the increasing number of window queries. We have three input configurations for each application type in our experimental settings. We select two inputs of an application run for the test dataset and place the remaining input for that application in the training dataset. We repeat the experiment with all possible scenarios in all application types. As seen in Figure 9, we start with an F1-score of 53%, and by querying an additional 2000 windows (357 samples), we reach 78%, which is very close to *F1-ALTD*. We reach perfect FAR after querying 50 windows. For AMR, it takes 200 windows to reach the perfect score.

5.4 Discussion of Deployment Scenarios

In this section, first, we discuss two potential scenarios to deploy the proposed approach in production. Then, we

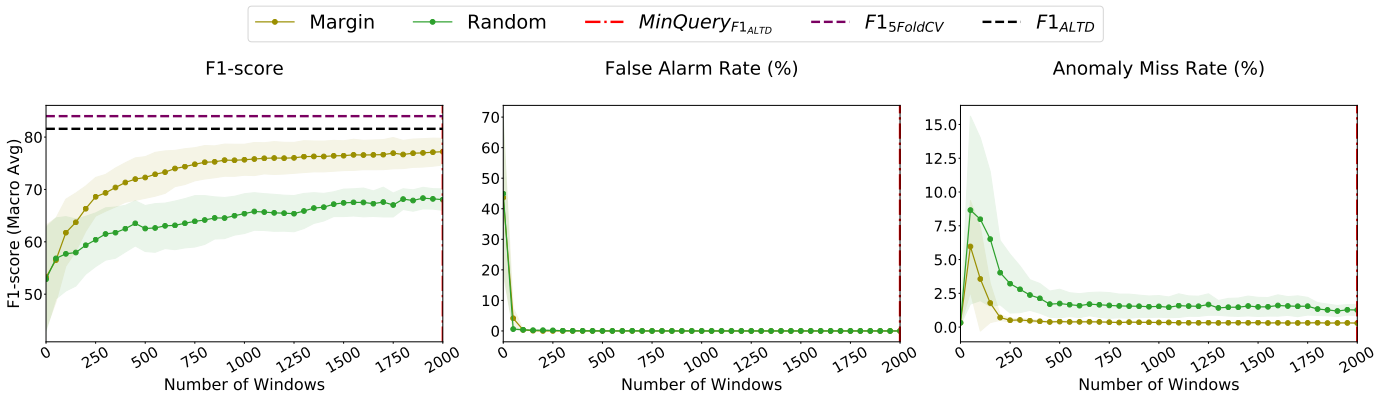


Fig. 9. The F1-scores, FARs, and AMRs when the training dataset consists of only the first input from each application and the test dataset includes the other two inputs. We repeat this experiment for all combinations. We reach a 0.77 F1-score by querying the label of an additional 2000 windows.

investigate the overhead of various deployment-related parameters for the selected deployment scenario.

Eclipse has a dedicated monitoring server, Shirley, to collect, process, and visualize data. Shirley comprises 16 compute nodes, each with 48 Intel Xeon Gold 6240R CPUs, 1.5 TB of memory, and 56 TB of NVMe storage. LDMS runs on all 1488 compute nodes on Eclipse, and the runtime telemetry data is aggregated into Shirley.

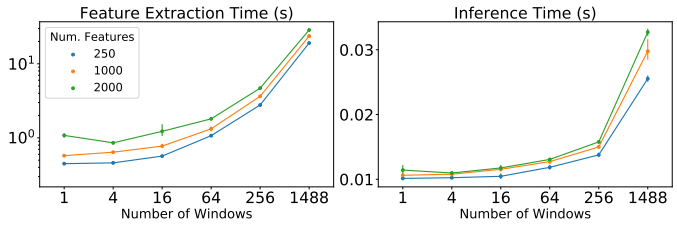


Fig. 10. Execution time analysis for feature extraction and inference stages for different numbers of windows. Using the model with 250 features, extracting features from 1488 windows takes 19.06, and generating predictions takes 0.03 seconds on average.

We investigate two scenarios to provide anomaly diagnosis results for the interval of interest. *Query-based Calculation* involves retrieving telemetry data for the selected time interval from the database and processing it through the *ML Pipeline*, which extracts features for each 60-second window and performs anomaly diagnosis. This approach does not store predictions, eliminating the need for storage and additional database management. However, latency may become a concern since the ML pipeline needs to generate millions of predictions when the job execution time is long or uses more than 500 nodes. *Stream-based Calculation* involves feeding the telemetry data to the ML pipeline as the data becomes available and storing the diagnosis results in another database. This predict-as-you-go approach significantly reduces the execution time overhead because the maximum number of windows processed equals the number of compute nodes in the worst case, which is 1488 for Eclipse. In terms of memory overhead, assuming every node is fully utilized for 24 hours and storing a 4-byte integer per prediction, the memory cost is approximately 35 MB daily without any compression or optimization strategies, which is also negligible. However, it requires managing another database. Although both approaches are feasible depending on system requirements, we focus on investigating the execution time overhead for the stream-based calculation approach as it has a lower overhead in the worst-case scenario.

TABLE 5
Time distribution of a single iteration during offline training.

	Query	Model Refit	Model Prediction	Other
Mean	6.56 s	1.05 s	26.4 s	6.1 s
Std	1.02 s	0.08 s	1.04 s	1.1 s

We investigate the execution time overhead of the best-performing model, RF, for feature extraction and inference stages across different numbers of windows. We measure the execution time for 1, 4, 16, 64, 256, and 1488 windows, spanning the maximum possible range. While we obtain the best performance with the model trained using 2000 features, we also utilize the same model with 250 and 1000 features to demonstrate how the execution time overhead scales with the number of features extracted. Both models exhibit less than a 2% performance drop in the F1-score for the same number of labeled samples. To select the optimal number of processes, we start by turning off the parallelization (n_jobs=0) and then systematically increase the number of processes to the maximum (n_jobs=48). Using all available cores for the feature extraction stage results in the fastest execution time in most cases. However, parallelization slows the process up to 10 times for the inference stage since the operation is not computationally intensive. As a result, we use all 48 cores for feature extraction and disable parallelization for inference. Figure 10 shows the time spent during the feature extraction and inference stages across different numbers of windows for models trained with different numbers of features. Extracting 250, 1000, and 2000 features from one window takes an average of 0.45, 0.57, and 1.08 seconds, respectively. For 1488 windows, extracting 250, 1000, and 2000 features takes 19.06, 23.64, and 28.5 seconds on average. In the worst case, inference time takes 0.03 seconds on average, which is negligible. We run all experiments on a single compute node, but it is possible to distribute 1488 windows to 16 nodes (i.e., processing 90

windows per node), which takes approximately 2 seconds.

For the offline training, we use a compute node with two fourteen-core 2.4 GHz Intel Xeon E5-2680v4 processors. A single iteration of batch mode active learning mainly includes querying 50 windows, refitting the model, and measuring the current performance. On average, one iteration takes 40 seconds on average. The exact distribution is reported in Table 5. To train the best performing model (Figure 3) with 1100 windows takes 880 seconds on average.

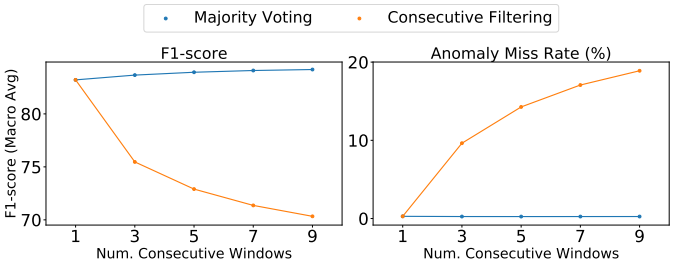


Fig. 11. Change in F1-score and AMR for two different filtering techniques for different numbers of consecutive windows. Majority voting increases the F1-score by 3% while keeping a perfect AMR and FAR.

We also explore two different runtime filtering methods to investigate whether we can improve FAR and F1 scores. The first method, named *majority voting*, involves replacing the original class prediction of the model with the most frequent class prediction across C consecutive windows. The second method, named *consecutive filtering*, retains the original classification prediction if a consistent class appears in C consecutive windows. If not, the prediction is replaced with a label indicating a healthy state. While this method effectively reduces the FAR by setting a confidence threshold for anomalous predictions, it often leads to a higher AMR, as some anomalous predictions get replaced with healthy labels.

To assess the impact of these filtering methods, we calculate the F1-score, FAR, and AMR by comparing each window's predicted class to its corresponding ground truth value. Figure 11 shows the change in F1-score and AMR with increasing C consecutive windows. In the majority voting case, we observe a slight increase in the F1-score while having perfect AMR. Unlike consecutive filtering, we observe a lower F1-score with higher C values due to increased AMR. It is important to note that both methods achieve a perfect FAR, but due to a higher F1-score, we find the majority voting method more feasible.

6 Conclusion and Future Work

The fluctuation of application performance in HPC systems not only impairs user satisfaction but also diminishes resource utilization efficiency and squanders computing power. As the magnitude and intricacy of large-scale systems continue to expand, telemetry data-based automated analytics are becoming increasingly necessary for reliable and efficient service. Although active learning frameworks have become prevalent in different domains where labeled data is scarce, there has been a lack of utilization of such frameworks for anomaly diagnosis. We present an active learning framework to identify previously encountered performance anomalies at runtime. We evaluate our framework using telemetry data collected from a production HPC system and show that we use 16 times fewer labeled samples compared to a supervised baseline utilizing the entire active learning training dataset, even in the presence of previously unseen applications and inputs in the test dataset.

In this paper, our framework operates under the assumption that annotators consistently provide accurate labels for selected windows or samples. This assumption, while effectively establishing an upper limit for the performance of our approach, presents an open problem regarding its practical applicability in production environments. One potential solution is to assess the impact of the contamination ratio, defined as the proportion of incorrect labels relative to the total number of labels, on model performance across diverse experimental settings [46]. This examination could lead to the development of more robust query strategies that are better equipped to handle incorrect labels. Another open problem arises from our reliance on domain expertise or established systematic approaches for annotation. The development of human-guided frameworks that incorporate domain-specific heuristics and rules to improve the annotation process presents a significant area for future exploration.

Acknowledgments

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] 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 Int. Conf. on Cluster Computing (CLUSTER)*, 2020, pp. 162–168.

[2] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "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.

[3] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs," in *IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[4] D. Skinner and W. Kramer, "Understanding the Causes of Performance Variability in HPC Workloads," in *IEEE Workload Characterization Symposium*, 2005, pp. 137–149.

[5] A. Das, F. Mueller, and B. Rountree, "Systemic Assessment of Node Failures in HPC Production Platforms," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 267–276.

[6] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson, "Toward Rapid Understanding of Production HPC Applications and Systems," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, 2015, pp. 464–473.

[7] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing Failures in Exascale Computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[8] J. Brandt, F. Chen, V. De Sapio, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong, "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.

[9] W. Barth, *Nagios: System and network monitoring*. No Starch Press, 2008.

[10] 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 *IEEE Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[11] A. Borghesi, A. Burrello, and A. Bartolini, "Examon-X: A Predictive Maintenance Framework for Automatic Monitoring in Industrial IoT Systems," *IEEE Internet of Things Journal*, 2021.

[12] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "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.

[13] A. Borghesi, A. Libri, L. Benini, and A. Bartolini, "Online Anomaly Detection in HPC Systems," in *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2019, pp. 229–233.

[14] 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.

[15] S. R. Kumar, P. D. B, and G. R. G, "Supervised Machine Learning-based Anomaly Detection and Diagnosis in Grid Connected Photovoltaic Systems." EAI, 12 2021.

[16] M. Hosseinzadeh, A. M. Rahmani, B. Vo, M. Bidaki, M. Masdari, and M. Zangakani, "Improving Security Using SVM-based Anomaly Detection: Issues and Challenges," *Soft Computing*, vol. 25, no. 4, pp. 3195–3223, 2021.

[17] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Anomaly Detection Using Autoencoders in High Performance Computing Systems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 9428–9433.

[18] B. Aksar, E. Sencan, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, B. Kulis, and A. K. Coskun, "Albadross: Active Learning-based Anomaly Diagnosis for Production HPC Systems," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, 2022, pp. 369–380.

[19] B. Settles, "Active Learning Literature Survey," 2009.

[20] T. Huang, P. Chen, and R. Li, "A Semi-Supervised VAE-based Active Anomaly Detection Framework in Multivariate Time Series for Online Systems," in *Proceedings of the ACM Web Conference*, 2022, pp. 1797–1806.

[21] W. Wang, P. Chen, Y. Xu, and Z. He, "Active-Mtsad: Multivariate Time Series Anomaly Detection with Active Learning," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 263–274.

[22] Z. Li, Y. Zhao, Y. Geng, Z. Zhao, H. Wang, W. Chen, H. Jiang, A. Vaidya, L. Su, and D. Pei, "Situation-aware Multivariate Time Series Anomaly Detection Through Active Learning and Contrast VAE-based Models in Large Distributed Systems," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 9, pp. 2746–2765, 2022.

[23] S. A. Khowaja and P. Khuwaja, "Q-Learning and LSTM-based Deep Active Learning Strategy for Malware Defense in Industrial IoT applications," *Multimedia Tools and Applications*, vol. 80, no. 10, pp. 14 637–14 663, 2021.

[24] 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.

[25] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing Performance Variations in HPC Applications Using Machine Learning," in *International supercomputing conference*. Springer, 2017, pp. 355–373.

[26] E. Ates, O. Tuncer, A. Turk, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Taxonomist: Application Detection Through Rich Monitoring Data," in *European Conference on Parallel Processing*. Springer, 2018, pp. 92–105.

[27] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller, "Data Mining-based Analysis of HPC Center Operations," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, 2017, pp. 766–773.

[28] 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, pp. 634–644, 2019.

[29] 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.

[30] 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.

[31] T. Schreiber and A. Schmitz, "Discrimination Power of Measures for Nonlinearity in a Time Series," *Physical Review E*, vol. 55, no. 5, p. 5443, 1997.

[32] T. P. Hill, "A Statistical Derivation of the Significant-Digit Law," *Statistical science*, pp. 354–363, 1995.

[33] C. D. Manning, *Introduction to information retrieval*. Syngress Publishing,, 2008.

[34] B. Settles, "Active Learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 6, no. 1, pp. 1–114, 2012.

[35] J. M. Joyce, "Kullback-Leibler Divergence," in *International encyclopedia of statistical science*. Springer, 2011, pp. 720–722.

[36] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[37] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme Scaling and Performance Across Diverse Architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

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

[39] "Exascale Proxy Applications." [Online]. Available: https://proxyapps.exascaleproject.org

[40] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sîrbu, A. Bartolini, and A. Borghesi, "Finj: A Fault Injection Tool for HPC Systems," in *Euro-Par 2018: Parallel Processing Workshops*, G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R. R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J. D. Garcia Sanchez, and S. L. Scott, Eds. Cham: Springer International Publishing, 2019, pp. 800–812.

[41] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "GPCNeT: Designing a Benchmark Suite for Inducing and Measuring Contention in HPC Networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[42] R. Han, O. R. Gatla, M. Zheng, J. Cao, D. Zhang, D. Dai, Y. Chen, and J. Cook, "A Study of Failure Recovery and Logging of High-Performance Parallel File Systems," *ACM Trans. Storage*, vol. 18, no. 2, apr 2022. [Online]. Available: https://doi.org/10.1145/3483447

[43] E. Ates, Y. Zhang, B. Aksar, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "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.

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

[45] B. Aksar, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "E2EWatch: An End-to-End Anomaly Diagnosis Framework for Production HPC Systems," in *European Conference on Parallel Processing*. Springer, 2021, pp. 70–85.

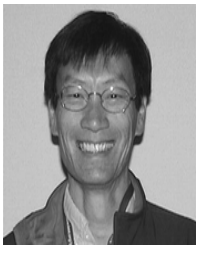
[46] Z. Liu, Z. Wang, Y. Yao, L. Zhang, and L. Shao, "Deep Active Learning with Contaminated Tags for Image Aesthetics Assessment," *IEEE Transactions on Image Processing*, 2018.

Burak Aksar is a Ph.D. student in the Department of Electrical and Computer Engineering at Boston University. He received his B.S. degree in Electronics Engineering from Sabanci University, Istanbul, Turkey. His research interests are applied machine learning & explainable AI techniques to improve the performance of large-scale computing systems. He has completed successful internships at IBM AI Research and Sandia National Labs.

Efe Sencan is a Ph.D. student in the Department of Electrical and Computer Engineering at Boston University. He received his B.S. degree in Computer Science and Engineering with a minor in Mathematics from Sabanci University, Istanbul, Turkey. His research interests are applied machine learning to improve the performance and efficiency of HPC systems.

Benjamin Schwaller is an R&D member of Sandia National Laboratories' High-Performance Computing Development division. He earned his Bachelor's and Master's degrees in Electrical Engineering from the University of Florida and the University of Pittsburgh, respectively. He was a research assistant at the NSF Center for Space, High-performance, and Resilient Computing (SHREC) from 2015 to 2018. His research interests include hardware performance optimization and supercomputing development.

Omar Aaziz is an R&D Computer Scientist at Sandia National Laboratories. He received an M.S. in Computer Science from Baghdad University in 2003 and a Ph.D. from New Mexico State University in 2018. He worked as an instructor at Baghdad University in Iraq from 2003 to 2009. His research interests are High-Performance Computing, specifically in dynamic analysis, application performance, and runtime monitoring. In the HPC arena, he used several statistical and machine learning modules to improve scientific application performance. His current research focuses on studying the increase of the overall utility of expensive HPC resources in a scientific application and analyzing the behavior patterns of parallel applications using statistical and numerical techniques.



Vitus J. Leung is a principal member of the Technical Staff at Sandia National Laboratories, where he leads research in distributed memory resource management. He has won R&D 100, US Patent, and Federal-Laboratory-Consortium Excellence-in-Technology-Transfer Awards for work in this area. He is a senior member of the ACM and IEEE and has been a member of the Technical Staff at Bell Laboratories in Holmdel, New Jersey, and a Regents Dissertation fellow at the University of California.

Jim Brandt is a distinguished member of the Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico, where he leads research in HPC monitoring and analysis.

Brian Kulis is an associate professor at Boston University, with appointments in the Department of Electrical and Computer Engineering, the Department of Computer Science, the Faculty of Computing and Data Sciences, and the Division of Systems Engineering. He also is an Amazon Scholar, working with the Alexa team. Before joining Boston University, he was an assistant professor in Computer Science and Statistics at Ohio State University, and before that was a postdoctoral fellow at UC Berkeley EECS. His research focuses on machine learning, statistics, computer vision, and large-scale optimization. He obtained his Ph.D. in computer science from the University of Texas in 2008 and his B.A. from Cornell University in computer science and mathematics in 2003. He has won three best paper awards for his research at top-tier conferences (International Conference on Machine Learning and IEEE Conference on Computer Vision and Pattern Recognition). He also received an NSF CAREER Award in 2015, an MCD graduate fellowship from the University of Texas (2003-2007), and an Award of Excellence from the College of Natural Sciences at the University of Texas.

Manuel Egele is an associate professor with the Electrical and Computer Engineering Department, Boston University (BU). He is the head of the Boston University Security Lab, where his research focuses on the practical security of commodity and mobile systems. He is a member of the IEEE and the ACM.

Ayse K. Coskun is a full professor in the Electrical and Computer Engineering Department at Boston University and the director of the Center for Information and Systems Engineering. Her research focuses broadly on design automation and computer systems, particularly focusing on energy efficiency, thermal challenges, and using analytics for intelligent system management. Coskun led multi-institutional projects, authored over 120 technical papers, taught classes on computer systems and software, and delivered many invited talks and tutorials. Her research outcomes are widely recognized and culminated in several technical awards, including the IEEE CEDA Ernest Kuh Early Career Award and an IBM Faculty Award. Coskun received her Ph.D. degree in Computer Science and Engineering from UC San Diego.