Prodigy: Towards Unsupervised Anomaly Detection in Production HPC Systems

Burak Aksar
baksar@bu.edu
Boston University
Boston, MA, USA

Efe Sencan
esencan@bu.edu
Boston University
Boston, MA, USA

Benjamin Schwaller
bschwal@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Omar Aaziz*
omarraad.aaziz@gmail.com
Sandia National Laboratories
Albuquerque, NM, USA

Vitus J. Leung
vjleung@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Jim Brandt
brandt@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Brian Kulis
bkulis@bu.edu
Boston University
Boston, MA, USA

Manuel Egele
megele@bu.edu
Boston University
Boston, MA, USA

Ayse K. Coskun
acoskun@bu.edu
Boston University
Boston, MA, USA

ABSTRACT

Performance variations caused by *anomalies* in modern High Performance Computing (HPC) systems lead to decreased efficiency, impaired application performance, and increased operational costs. While machine learning (ML)-based frameworks for automated anomaly detection (often based on time series telemetry data) are gaining popularity in the literature, practical deployment challenges are often overlooked. Some ML-based frameworks require extensive customization, while others need a rich set of labeled samples, none of which are feasible for a production HPC system.

This paper introduces a variational autoencoder-based anomaly detection framework, *Prodigy*, that outperforms the state-of-the-art alternatives by achieving a 0.95 F1-score when detecting performance anomalies. The paper also provides a real system implementation of Prodigy that enables easy integration with monitoring frameworks and rapid deployment. We deploy Prodigy on a production HPC system and demonstrate 88% accuracy in detecting anomalies. Prodigy involves an interface to provide job- and node-level analysis and explanations for anomaly predictions.

CCS CONCEPTS

• **Computing methodologies** → **Unsupervised learning**; **Machine learning**; • **General and reference** → **Performance**; • **Software and its engineering**;

KEYWORDS

High Performance Computing, Anomaly Detection, Machine Learning, Deployment

ACM Reference Format:
Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K. Coskun. 2023. Prodigy: Towards Unsupervised Anomaly Detection in Production HPC Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607076

1 INTRODUCTION

Modern High Performance Computing (HPC) systems are crucial for many fields, such as drug discovery, climate modeling, nuclear physics, and financial modeling. These systems have become more complex and parallel following the exponential growth in data and computing power. With the move towards exascale computing along with large-scale systems hitting more severe limitations in power consumption, newer HPC systems are being designed with higher degrees of resource sharing, which, in turn, reduces performance predictability. According to Chunduri et al. [18], the run-to-run variability in an application's execution time may exceed 70%, even if the same input deck is used. Similarly, other studies [27, 45, 51] show that performance *anomalies* can cause an increase in a job's execution time by over 100%. Performance anomalies occur due to various factors, including network contention, [13], hardware-related problems [29], memory issues [2], shared resource contention [21], and CPU-related problems [46]. While performance anomalies that cause system crashes are typically easier to detect, those that result in degraded system efficiency can be more challenging to identify since they do not always lead to node or system failures. Therefore, this paper focuses on the more challenging problem of detecting performance anomalies that do not result in system crashes but lead to degraded system efficiency.

An HPC system today typically employs one or more monitoring frameworks to gather information on resource utilization

*This work has been completed while the author was at Sandia National Laboratories.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607076

during application runs. Often, these frameworks collect telemetry data in the form of multivariate time series data gathered via performance counters at various levels of the system, system logs, and traces, which can be analyzed to detect anomalous behavior. However, manual analysis is impractical due to the massive volume of telemetry data generated by HPC systems (i.e., billions of data points per day). Moreover, each HPC application may exhibit unique characteristics, requiring domain expertise to differentiate healthy and anomalous states. Therefore, automated performance analysis techniques, such as machine learning (ML)-based frameworks, are gaining attention in the HPC community for effectively detecting and diagnosing performance anomalies in large-scale computing systems [5, 15, 26, 31, 49]. Recent work has designed ML-based frameworks for detecting anomalies in supervised learning settings [3, 19, 49]. Although these supervised frameworks achieve high anomaly detection accuracy, their performance is closely tied to the amount and quality of labeled data. However, the overwhelming majority of the telemetry data collected from HPC systems are not labeled as healthy or anomalous due to the immense scale of the data ingested and the wide range of domain expertise needed. To alleviate this problem, recent work has introduced semi-supervised [4, 5, 14] and unsupervised frameworks [31, 34] that operate with limited or only healthy labeled data. These frameworks are better suited for production systems compared to fully-supervised ML frameworks; however, their applicability is also limited as existing frameworks deploy compute-node-specific models [14] or focus on detecting and forecasting node failures [31] rather than performance anomalies, which can often be more difficult to detect.

To address the open problems of *limited labeled data* and *arduous deployment processes*, we introduce *Prodigy*[1], an unsupervised[2] anomaly detection framework that detects performance anomalies on compute nodes. We also design a simple and flexible architecture that allows for easy deployment of *Prodigy* and demonstrate its effectiveness on a 1488-node production HPC system. Our framework assumes having access to healthy labeled *samples*[3] during the training phase to learn the healthy application run characteristics. This is a reasonable assumption, considering anomalies occur infrequently in production HPC systems. Therefore, most of the time, the system is expected to operate within "normal" ranges of performance. We evaluate the anomaly detection performance of our framework on a production HPC system and also using data collected from an HPC testbed. Our specific contributions are as follows:

- Design of *Prodigy*, a variational autoencoder (VAE)-based framework, that detects performance anomalies on compute nodes using multivariate time series telemetry data. Our framework achieves 0.95 and 0.88 F1-scores on a production system (Eclipse) and an HPC testbed (Volta) telemetry datasets, respectively.
- Application of a state-of-the-art counterfactual explainability framework CoMTE [7] to the anomaly detection problem, which sheds light on the decisions of black-box models and helps HPC administrators and users understand the root cause of anomalies.
- Design of a simple and customizable software architecture for *Prodigy* and demonstrating the effectiveness of the *Prodigy* framework by deploying on a production HPC system with 1488 compute nodes. Our framework achieves an 88% accuracy in detecting real-world performance anomalies.

The rest of the paper starts with an overview of related work, followed by introducing our anomaly detection framework in Section 3. Next, we discuss the software architecture and deployment of *Prodigy* in Section 4. The experimental methodology is explained in Section 5, while the results are presented in Section 6. The paper concludes in Section 7.

2 RELATED WORK

Anomaly detection in multivariate time series data is a crucial area of research with a multitude of practical applications. In this section, we provide an overview of recent developments in anomaly detection methods in the context of large-scale computing systems. Additionally, we examine the deployment of ML-based anomaly detection frameworks for this purpose.

2.1 Anomaly Detection in HPC Systems

In the past decade, various supervised ML frameworks have been designed for detecting anomalies. In a supervised setting, the model is trained using labeled data that includes healthy and anomalous samples. The idea is to teach the model to recognize the patterns and characteristics that distinguish healthy data from anomalous data, then use this knowledge to classify unseen data accurately. For instance, Tuncer et al. [49] introduce a novel framework for diagnosing performance anomalies. Their approach involves statistical feature extraction and feature selection techniques to process data collected from an HPC system. They then train a decision tree-based classifier to identify and classify different types of performance anomalies. Baseman et al. [11] introduce a framework for fault detection in supercomputers using a statistical method called classifier-adjusted density estimation. They employ a Random Forest classifier trained on both real and artificially generated data to predict performance anomalies. Klinkenberg et al. [26] monitor power consumption and network traffic data of applications without adding measurement overhead on compute nodes. They then use descriptive statistics and a supervised ML model that predicts imminent node failures caused by hardware or software issues with high accuracy.

In a semi-supervised setting, the model is only trained with a relatively small portion of the labeled and many unlabeled samples. Borghesi et al. [14] design a novel semi-supervised approach for anomaly detection in HPC systems using autoencoder neural networks [10]. Their approach requires data from normal system states and detects anomalous conditions (e.g., anomalies due to misconfigured CPU frequencies with respect to the current workload) without the need for many examples of anomalous states. Aksar et al. [5] build a semi-supervised anomaly diagnosis framework that

[1] Our implementation is available at: https://github.com/peaclab/Prodigy
[2] Even though *Prodigy* is based on unsupervised learning model (i.e., variational autoencoders), its training stage uses only healthy samples.
[3] A sample is a vector (1 x N features) generated by extracting features of multivariate telemetry data (Time x M metrics) collected from a compute node during an application run.

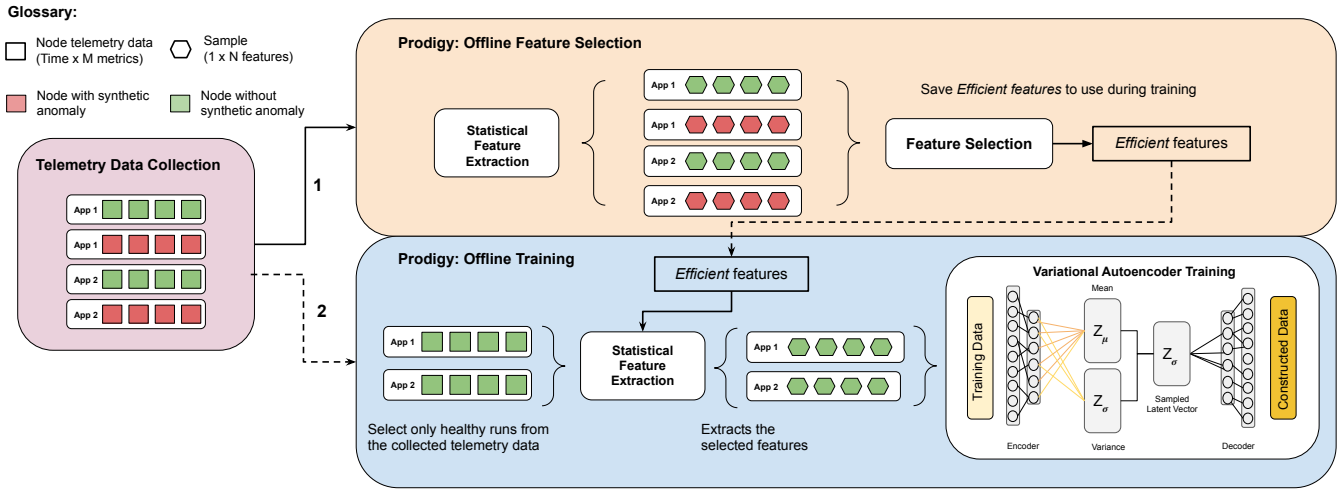

Figure 1: The high-level workflow of our framework, *Prodigy*. First, we extract statistical features from telemetry data collected from healthy and anomalous application runs. We then employ the Chi-square feature selection method to determine the most discriminative features (*efficient features*) that can separate healthy from anomalous samples. In the offline model training stage, we only use healthy application runs to train the model and determine a reconstruction error-based threshold to detect anomalies.

first uses autoencoders to learn the characteristics of healthy and anomalous data. They then use a supervised classifier to determine the anomaly types. However, their framework's performance may be limited as it solely relies on existing labeled samples. In their most recent work [4], authors design a novel active learning-based [44] anomaly diagnosis framework that significantly reduces the need for labeled samples during the training phase while achieving high classification performance.

In an unsupervised learning setting, the ML model is trained with unlabeled samples. This training setup aims to identify the hidden patterns and relationships in the data without any explicit guidance. State-of-the-art unsupervised learning frameworks employ deep neural networks for detecting anomalies in multivariate time series data. Audibert et al. [9] propose USAD, which is based on adversely trained autoencoders to isolate anomalies while having fast training time. Deng and Hooi [20] use Graph Neural Networks [40] for identifying anomalies via attention-based forecasting and deviation scoring. TranAD [48] is a deep transformer network-based anomaly detection model that utilizes attention-based sequence encoders with a focus on score-based self-conditioning and adversarial training of autoencoders.

Regarding anomaly detection frameworks in the HPC domain, Molan et al. [31] design a recurrent autoencoder model that detects anomalies in compute nodes. Their framework captures the temporal dependencies in the time-series data by including long short-term memory [50] cells in the model architecture and achieves high anomaly detection accuracy. Ozer et al. [34] employ Bayesian Gaussian mixture models [39] to analyze HPC monitoring data. Their method enables the extraction of statistical information about the behavior of individual components within the HPC system through Gaussian distributions.

2.2 Deployment

Operational Data Analytics (ODA) solutions [16, 32, 33] are becoming increasingly popular for their ability to provide real-time system insights to users and system administrators. One important application of ODA is detecting performance variations and understanding the root cause of the anomalies in compute nodes. ML-based frameworks greatly enhance the ability of system administrators to identify and address potential issues before they cause significant problems by detecting and diagnosing anomalies in compute nodes. As such, the deployment of ML-based frameworks is a promising area of research that has the potential to improve the efficiency and effectiveness of ODA solutions significantly.

Molan et al. [31] design a node-specific autoencoder model and deploy it on a production HPC system with 980 nodes. They only focus on detecting anomalies that cause severe malfunctioning of a compute node (i.e., anomalies that cause a compute node failure). However, our focus is on detecting anomalies that do not result in node failures since these anomalies are generally harder to detect. Moreover, maintaining and configuring separate ML models for each compute node can be challenging as the size of the system scales to thousands of compute nodes. Therefore, having a generic model is a more feasible approach for the production system scenario.

Our framework differs from prior works in the following directions. We design a VAE-based anomaly detection framework that only requires healthy labeled samples during the training stage. We also demonstrate the effectiveness of our framework by deploying on a production HPC system with 1488 compute nodes and provide an interface that includes job- and node-level analysis as well as counterfactual explanations for anomalous predictions.

3 *PRODIGY*

Our primary goal is to identify whether any compute node within a system displays anomalous behavior that leads to performance variations. We are particularly interested in detecting anomalies that cause performance variability without resulting in program errors or premature termination, as such anomalies tend to be more difficult to detect.

Prodigy is a VAE-based unsupervised framework that detects performance anomalies on compute nodes. Figure 1 summarizes the high-level workflow. The telemetry data collection stage shows how we run controlled experiments to test the performance of *Prodigy* before deployment. Note that this stage is optional, and we share more guidance on the deployment pipeline in Section 4. We collect telemetry data from compute nodes while running applications with and without synthetic anomalies. The offline feature selection stage aims to select the most discriminative features that can separate healthy from anomalous samples. During the offline training stage, we only use telemetry data collected from compute nodes without synthetic anomalies. As anomalies are exceedingly rare, we assume all samples from this collected data are healthy. We extract the statistical features of the telemetry data and train a VAE to learn an unsupervised representation (encoding) of healthy samples. In the following sections, we provide details.

3.1 Feature Extraction

Feature extraction is a critical step in deriving meaningful information from raw telemetry time series data. We use an open-source toolkit known as TSFRESH [17], which computes 794 features for each metric based on 63 distinct time series characterization methods. Some features are descriptive statistics (e.g., min, max, mean, etc.) along with a more extensive and advanced set of features, including, but not limited to, approximate entropy, power spectral density, and variation coefficient.

3.2 Feature Selection

While feature extraction is essential to generate new features from raw telemetry data, not all the extracted features may be necessary or contribute significantly during training. Therefore, using a feature selection methodology is crucial to identify the most discriminative features and reduce dimensionality for faster training. To ensure the selection of features that effectively discriminate between healthy and anomalous samples, our approach utilizes a dataset comprising both anomalous and healthy samples only in this stage. We use the Chi-square [35] feature selection method, which measures the dependency between each feature and the class variable. Features with higher Chi-square values are considered more important for predicting the class variable.

3.3 Unsupervised Training

This stage aims to learn the characteristics of the "healthy" samples in an unsupervised setting using a VAE model. VAE is a generative neural network that learns a lower-dimensional representation of the input data and generates new samples similar to the input data. The overarching goal is to minimize the reconstruction error, which is a measure of how well the VAE can reconstruct the input data. During training, the VAE model requires only input data. It learns the underlying probability distribution of healthy samples without accessing class labels; hence, it is considered in the unsupervised learning category.

VAEs are a variant of the traditional autoencoder architecture, which consists of an encoder network that maps input data to a latent space and a decoder network that maps latent variables back to the input space. The key difference between VAEs and traditional autoencoders is that VAEs are designed to learn a continuous and smooth representation of the latent space. Unlike standard autoencoders, the latent space is regularized to have a simple and tractable distribution, which can be used to generate new samples as well as identify potential outliers/anomalies.

The latent space assumes a prior distribution, typically a standard normal distribution. The encoder network maps the input data to the parameters of a Gaussian distribution in the latent space, with a mean vector μ and diagonal covariance matrix Σ:

$$q_\phi(z|x) = \mathcal{N}(\mu(x;\phi), \Sigma(x;\phi)) \tag{1}$$

where x is the input data, ϕ are the parameters of the encoder network, and z represents the latent variables. The decoder network then maps these latent variables back to the input space, with a conditional distribution $p_\theta(x|z)$. To train a VAE, we need to maximize the evidence lower bound (ELBO) on the log-likelihood of the data, which is given by:

$$\log p_\theta(x) \geq \mathbb{E}q\phi(z|x)[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p(z)) \tag{2}$$

where $p(z)$ is the prior distribution on the latent space, and Kullback-Leibler (KL) divergence between the encoder distribution and the prior distribution is defined as:

$$D_{KL} = (q_\phi(z|x)||p(z)) \tag{3}$$

This objective function can be optimized using stochastic gradient descent, where the gradient of the ELBO with respect to the network parameters is estimated using backpropagation. To calculate gradients efficiently, we use the reparameterization trick. Instead of directly sampling from the Gaussian distribution in the latent space, we sample from a standard normal distribution ϵ and then transform it using the mean and covariance of the Gaussian distribution:

$$z = \mu + \Sigma^{1/2} \odot \epsilon \tag{4}$$

where $\odot$ denotes element-wise multiplication. After the training is completed, the initial step involves determining a threshold. This threshold is typically determined through various statistical calculations like the 99^{th} percentile or maximum value and represents the acceptable range to classify a sample as healthy. To determine the threshold, we measure the reconstruction error using mean absolute error for each sample in the training dataset (only healthy samples) and then set the threshold as the 99^{th} percentile. The process does not necessitate manual intervention, but one can experiment with different percentile values.

3.4 Anomaly Detection

During the evaluation stage, the VAE model reconstructs the input sample using the latent space. We then compute the reconstruction

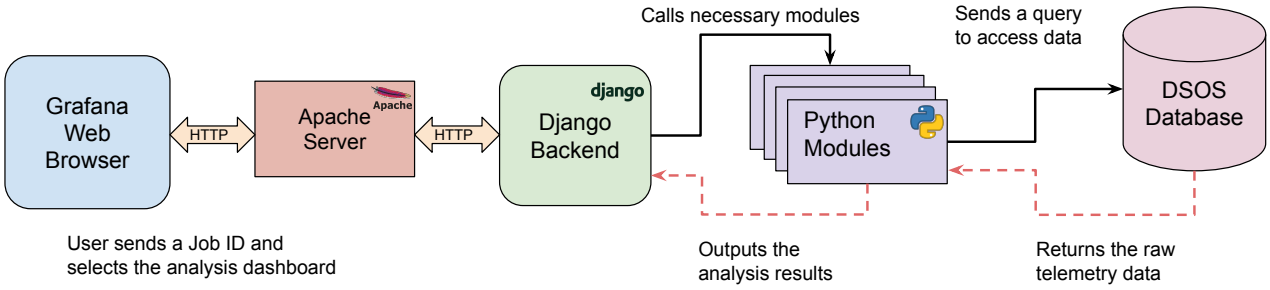

Figure 2: High-level flow of the analytics pipeline in Eclipse.

error by comparing the original input sample to its corresponding output. A lower reconstruction error indicates that the model can generate the sample with higher confidence, while a higher reconstruction error indicates that the sample is likely from a distribution not seen during training. To determine whether the new sample is anomalous, we compare its reconstruction error to the determined threshold. If the reconstruction error is below the threshold, we classify the sample as healthy. If the reconstruction error exceeds the threshold, we classify the sample as anomalous.

4 DEPLOYMENT PIPELINE

One of the main contributions of this paper is demonstrating a working prototype of *Prodigy* on a production HPC system. Our goals in deployment are enabling easy integration with monitoring frameworks and faster deployment. The software architecture we design is customizable (e.g., we support different feature extraction and selection strategies and various ML models) and requires only a monitoring framework and a backend server. This section details the end-to-end deployment flow.

4.1 Monitoring, Storage, and Analytics

HPC systems typically rely on monitoring frameworks that gather and analyze performance data from various subsystems to ensure efficient operation. Ganglia [30] and Lightweight Distributed Metric Service (LDMS) [1] are examples of monitoring frameworks commonly used in HPC systems. This paper uses LDMS since it is the default monitoring framework used on the production HPC system we work with for deploying and evaluating *Prodigy*.

LDMS collects telemetry data from various subsystems and performance counters at a sub-second granularity with minimal overhead in a distributed manner. To collect telemetry data, LDMS requires a sampler daemon, *ldmsd*, on each node of interest, where the sampler daemon samples metrics from specific subsystems such as memory from /proc/meminfo (*meminfo*), CPU from /proc/stat (*procstat*), and virtual memory from /proc/vmstat (*vmstat*). In this deployment, LDMS samples metrics from all compute nodes at 1Hz, and all data is aggregated and stored in a separate monitoring cluster.

The monitoring cluster hosts an analytics pipeline, which provides derived figures-of-merit about application- and system-related metrics, both at runtime and post-runtime, through a Grafana-based dashboard [43]. It also hosts a database technology called Distributed Scalable Object Storage (DSOS)[4], which is a scalable database designed specifically for continuous large-scale data ingestion and querying. The database has a variety of Python and bash APIs to query and change objects and to provide performant executions. Users can create dashboards based on their needs by calling different analysis modules on chosen LDMS metrics. For example, a system administrator can create a dashboard to find and display the jobs with the highest filesystem usage at a given time. On the other hand, an application developer can create a dashboard to investigate how specific metrics (e.g., *Active*, *MemAvailable*, *AnonPages*) change over application execution time. Figure 2 summarizes the analytics pipeline on the target system. First, a user inputs some parameters of interest, such as job ID and type of analysis dashboard they want to display (e.g., anomaly detection dashboard or CPU usage dashboard), through a Grafana interface. Parameters are sent through a Django-based backend server, and Python modules are called based on the selected analysis dashboard. The selected modules can query the DSOS database, perform the necessary data transformations, and return the output to the backend server. We create a new Python module to interface with *Prodigy* and a separate analysis dashboard to display anomaly detection results for a given job ID, as detailed in the next section.

4.2 Deploying *Prodigy*

This section describes how *Prodigy* is integrated into the existing analytics pipeline. First, we discuss offline model training and then describe anomaly detection and explainability components.

4.2.1 Offline Model Training. Figure 3 provides the high-level flow for data processing and model training. We provide a generic *DataGenerator* class that performs data preprocessing to generate necessary training and test datasets and labels[5] for ML models. DataGenerator expects raw telemetry data from available samplers (e.g., *meminfo*, *vmstat*, and *procstat*) and performs preprocessing for each job ID. Some collected metrics are accumulated, and we are interested in the relative change in those metrics, so we calculate the difference in each time step. We also apply linear interpolation to

[4]https://github.com/ovis-hpc/sos
[5]Label generation is not required for unsupervised models, but the provided class supports supervised and semi-supervised settings.

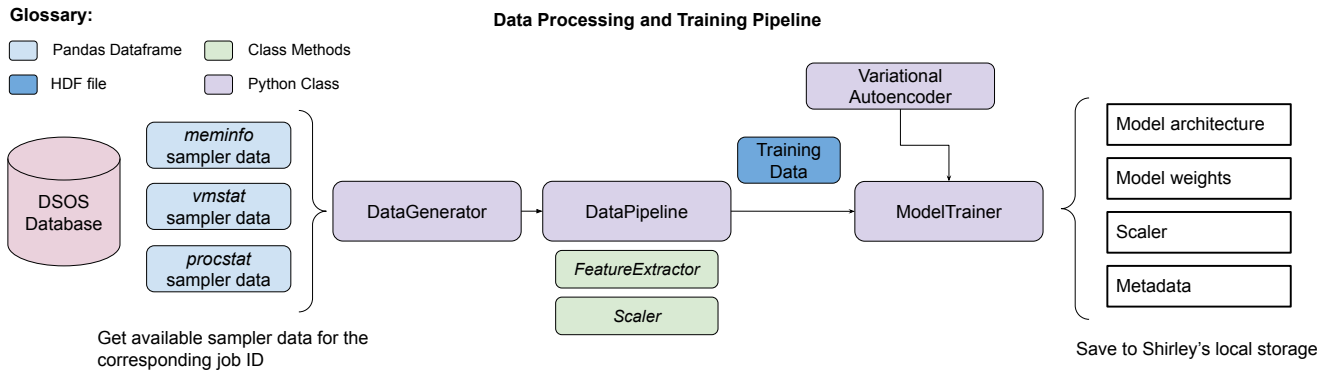

Figure 3: The high-level architecture of data processing and training pipelines.

every metric to fill in missing values, as some values may be lost during the collection stage. Then, we find common timestamps across different samplers and drop unused columns. The prepared datasets have three index columns: *job_id*, *component_id*, and *timestamp*, and the remaining columns correspond to metrics.

The *DataPipeline* class is designed to perform common operations for ML models before training and evaluation. The *Feature Extractor* method calculates a wide range of features such as descriptive statistics (e.g., mean, max, and standard deviation) and complex statistics such as C3 values [42] and Benford correlation [24]. The *Scaler* module fits the selected scaler (e.g., min-max scaler) to the training data, then transforms the test data (if available) and saves the scaler object.

The *ModelTrainer* is a generic class that trains the provided model and saves the necessary files to use the model in production. ModelTrainer expects the provided model as a subclass of *Keras Models*[6], which enables users to add their own models easily. We create a class for the proposed VAE architecture by following the implementation details discussed in Section 5.4. After the training process is completed based on provided hyperparameters, ModelTrainer saves the trained model, weights, scaler, and deployment metadata (e.g., the columns of training data and extracted features) in the specified output directory.

4.3 Anomaly Detection

Figure 4 shows the prediction pipeline. When a user selects the anomaly detection dashboard, the backend server calls the required module. First, DataGenerator queries the sampler data from the DSOS database, applies preprocessing, and then feeds it to DataPipeline. The DataPipeline module extracts the same features used during training, scales the extracted features, and passes them to AnomalyDetector. AnomalyDetector loads the scaler, the deployment metadata, and the pre-trained model. Finally, we provide a binary prediction for each compute node (*component_id*) that exists in the provided *job_id*, and display the results through a new Grafana analysis dashboard.

[6]https://keras.io/api/models/model/

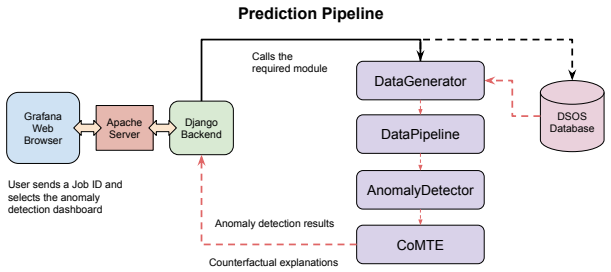

Figure 4: A user provides a job ID and selects the anomaly detection dashboard. For each compute node associated with the job ID, we provide binary anomaly detection results. We also provide CoMTE explanations for anomalous predictions.

4.4 Explainability

The motivation behind using an explainability framework after detecting anomalies is to understand better why the model identified certain compute nodes as anomalous. Explainability frameworks can help provide insights into the model's decision-making process, increase transparency, and improve trust in the model's output. We investigate popular feature-based explainability methods (e.g., LIME [38], SHAP [28]) and time series-specific explainability techniques. However, we ultimately decide to use the open-source implementation of "CoMTE: Counterfactual Explanations for Multivariate Time Series" [7]. The first reason is that feature-based explainability methods assign positive and negative scores to each feature, but they cannot determine the minimum number of features required for an explanation. This makes them infeasible for high-dimensional datasets. Second, other time series-specific explainability techniques are either limited to univariate time series [22, 41] or based on specific model architectures [6, 25]. In contrast, CoMTE can explain the predictions of multivariate time series models with any architecture. CoMTE formulates the problem of explaining time series via counterfactuals as follows. First, given a sample classified as anomalous that requires explanation, the goal is to identify (1) the distractor, which is a training set sample classified as healthy, and (2) the minimum set of time series (i.e., metrics) to

be replaced from the distractor sample, resulting in a change of the classification label to the healthy class.

Assume that the predictions for *job_id* 123 indicate that *component_id* 66 is healthy, whereas *component_id* 12 is anomalous. To interpret the anomalous predictions, CoMTE is initialized using the trained model and the training dataset. For example, the explanation is composed of *pgrotated* and *pginodesteal* metrics. It also informs us that the sample would be classified as healthy if the *pgrotated* metric were lower and the *pginodesteal* metric were higher. This explanation can help to identify the root cause of any anomalies and determine the appropriate actions to take to improve system performance.

5 EXPERIMENTAL METHODOLOGY

This section provides a detailed overview of the target HPC production system, applications, and synthetic anomalies used during experimental scenarios. We also provide the implementation details and design choices.

5.1 HPC Systems and Monitoring Server

Volta is a testbed supercomputer at Sandia National Laboratories with 52 computing nodes connected in 13 switches, each with four nodes. Each node has 64GB of memory and two sockets equipped with an Intel Xeon E5-2695 v2 CPU with 12 two-way hyper-threaded cores.

Eclipse is located at Sandia National Laboratories and consists of 1488 compute nodes, each equipped with 128GB of memory and two sockets. Each socket contains 18 E5-2695 v4 CPU cores with 2-way hyperthreading, providing substantial computational power for scientific and engineering applications. Shirley is a dedicated monitoring server for Eclipse and is responsible for collecting, processing, and displaying analysis results to the end users. It comprises 16 compute nodes, each with 48 Intel Xeon Gold 6240R CPUs, 1.5 TB of memory, and 56 TB of NVMe storage [43]. *ldmsd* samplers collect telemetry data from Eclipse compute nodes, and the sampled data is aggregated into Shirley every second. Currently, the resulting data is approximately 10 TB per day.

5.2 Applications and Synthetic Anomalies

Before deploying an ML model to production, evaluating its performance with a ground truth dataset is a common practice. However, acquiring such a dataset from production HPC systems poses significant challenges due to the infrequency of anomalies and the complexity of the collected telemetry data, which makes it difficult to label. To overcome this challenge, first, we select some common HPC applications (Table 1) and build them on the target system. We prepare three different input decks for each application to run on 4, 8, and 16 nodes for 20-45 minutes.

After the build, we run the listed applications with and without synthetic anomalies to collect ground truth labels. We utilize synthetic anomalies provided by the High Performance Anomaly Suite (HPAS), an open-source tool that generates anomalies designed to create contention across various subsystems, such as memory, network, and I/O [8]. We use the following anomalies: *memleak*, which simulates memory leakage by allocating an array of characters without storing the addresses; *membw*, which mimics memory bandwidth contention, preventing data from being loaded into the cache; *cpuoccupy*, which simulates excessive CPU utilization; and *cachecopy*, which mimics cache contention by swapping the contents of two arrays repeatedly for a specific cache level, such as the L3 cache. We also conduct experiments involving I/O- and network-related anomalies. Due to significant contention caused by I/O-related anomalies, system administrators terminated the runs. The network anomaly generates contention only when applications are run on two compute nodes; hence, it is not incorporated into our experiments. The configuration details of injected anomalies are provided in Table 2.

Table 1: The list of applications we run on Eclipse and Volta for data collection.

Eclipse		
	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
Volta		
	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

Table 2: A list of performance anomalies used in our experiments and their configurations.

Anomaly type	Configuration
cpuoccupy	-u 100%, 80%
cachecopy	-c L1,-m 1 / -c L2 -m 2
membw	-s 4K, 8K, 32K
memleak	-s 1M, -p 0.2 / -s 3M -p 0.4 / -s 10M -p 1

5.3 Baseline Models

To evaluate the effectiveness of our framework, we compare it against several baselines, including both deep learning and traditional anomaly detection methods. We do not experiment with non-ML-based methods (e.g., ARIMA [37] or exponential smoothing [36]) because they are typically designed for forecasting future values based on historical data and may struggle to effectively capture complex patterns and irregularities when non-linear trends

are present. One of the deep learning baselines we implement is USAD [9]. USAD employs a 2-stage autoencoder for training, where each autoencoder is trained to reconstruct input data. Then, adversarial training is used to make one autoencoder learn to distinguish between real data and reconstructions from the other autoencoder, while the other autoencoder learns to produce more realistic reconstructions to fool the discriminator. During inference, these dual-purpose autoencoders are used to calculate an anomaly score based on the reconstruction errors.

Among the traditional unsupervised learning algorithms, K-means clustering [23] is a popular method for grouping data points into clusters based on their similarity. However, this method may not be effective in detecting anomalies in high dimensional datasets or when the clusters are non-spherical [47]. Therefore, we implement Local Outlier Factor (LOF) as an alternative to K-means clustering. LOF calculates the density of each data point based on its local neighborhood, which takes into account the distribution of the data and is a more informative measure of similarity in high-dimensional spaces. Specifically, LOF calculates a score for each data point that indicates how far it is from its neighbors in terms of density. Points that have significantly lower densities than their neighbors are considered anomalies.

Another popular baseline we implement is Isolation Forest (IF), which can scale to large datasets while maintaining a high and robust anomaly detection performance. IF creates an ensemble of random trees that isolate anomalies in the data based on how quickly they are isolated from the rest of the data points. The algorithm randomly selects a feature and splits the data at a random value within the range of the feature. By repeating this process, the algorithm builds a tree structure that isolates the anomalies, as they require fewer splits to be separated from the rest of the data. The anomaly score is computed by averaging the number of splits required to isolate a data point across all the trees in the ensemble. Data points with lower scores are considered anomalies.

We also implement two heuristic baselines for comparison: Random Prediction and Majority Label Prediction. In Random Prediction, we randomly select the prediction output, while in Majority Label Prediction, we predict the label of each sample based on the majority label in the test dataset. Majority Label Prediction acts as a fundamental benchmark or starting point for classification tasks. If an ML model fails to surpass its performance, it indicates that the model is not providing any substantial improvement. Even though the majority of runs are expected not to be anomalous, our test dataset for Volta has 10%, whereas Eclipse has a 90% anomaly ratio. So in this context, Majority Label Prediction gives a better idea in terms of the generalization of the model. We provide more details on implementing the baselines in the next section.

5.4 Implementation Details

This section provides the implementation details for data collection, preprocessing, ML models, and CoMTE.

5.4.1 Data Collection and Preprocessing. The telemetry data has T timestamps, each belonging to the corresponding application run, and M metrics. We collect 806 metrics per second from every compute node using *meminfo*, *vmstat*, and *procstat* samplers. However, we exclude the *per-core* metrics (such as *per_core_cpu_enabled8*, *per_core_guest8*, *per_core_irq8*, etc.) because we observe significant fluctuations in core-level metrics for the same application run with the same input deck. This is primarily because the operating system dynamically allocates cores. In contrast, node-level aggregate CPU metrics exhibit greater stability and consistency. As a result, we end up with 156 metrics in total. In the data preprocessing stage, we eliminate the first and last 60 seconds as they mostly correspond to the initialization and termination phases, where some metrics may deviate significantly from expected values. We perform linear interpolation on each time series to fill in missing values. Additionally, we calculate the difference between each time step to capture the relative change in some *procstat*-related metrics as they are accumulated as raw values.

Table 3: Optimal hyperparameters are shown with a star (*) for each model.

Model	Hyperparameter Space
Prodigy	Learning rate: 1e-5, 1e-4*, 1e-3, 1e-2
	Batch Size: 32, 64, 128, 256*
	Num. Epochs: 400, 800, 1200, 2400*, 3000, 6000
USAD	Batch Size: 32, 64, 128, 256*
	Num. Epochs: 50, 100*, 200, 400
	Hidden Layer Sizes: 100, 200*, 400
	Alpha & Beta: 0.1, 0.5*, 1

5.4.2 Train-Test Split. After we run the applications and anomalies described in Section 5.2, we collect 24,566 (6,325 healthy) and 20,915 (18,980 healthy) samples from Eclipse and Volta systems, respectively. To create our training and test datasets, we split (20-80%) the data while maintaining the distribution of both normal and anomalous samples. The training and test dataset remains the same across all experiments. We also set the anomaly ratio (i.e., the number of anomalous samples divided by the total number of samples) as 10% in our training dataset. We determine this ratio by observing the outlier application runs in Eclipse. The outlier behavior is defined as runs with execution times of 1.5 interquartile range (IQR) below Q1 or 1.5 IQR above Q3. We observe that the outlier ratio ranged between 2-7%. As a result, we cap the anomaly ratio to 10%.

5.4.3 Feature Selection. The Chi-square feature selection method measures the statistical significance between a feature and the class variable by computing the Chi-square statistic; hence, it requires at least two classes (or labels) in the dataset. During our feature selection process, we use healthy and anomalous samples to determine the most discriminative features. Considering this, our feature selection process requires minimal supervision (i.e., 24 and 55 anomalous samples in total for Eclipse and Volta datasets, respectively). We experiment with the top 250, 500, 1000, and 2000 features, and our model performs best on the test set with 2000 features.

5.4.4 ML Models and CoMTE. We conduct a grid search to identify the optimal hyperparameters (e.g., learning rate, batch size, number of epochs, and the number of hidden layers) for each ML model. Table 3 shows the best values for hyperparameters of *Prodigy* and USAD. During the training process of *Prodigy* and USAD, we remove the anomalous samples in the training dataset and then divide the remaining data into additional training and validation sets (80-20%).

In USAD implementation, unlike the original paper, we do not divide the time series into windows; instead, we directly extract and select features from raw telemetry data. After completing the training process, we use the validation set to determine the optimal anomaly threshold for making predictions. We iterate through possible values between 0 and 1 with 0.001 increments and select the threshold that results in the highest F1-score in the test dataset. We report the F1-score in the larger part using the determined threshold.

For IF and LOF, we use their scikit-learn implementations. During their training process, we keep anomalous samples in the training dataset as they can work with healthy and anomalous samples. For both methods, we set the contamination ratio to 10% due to the anomaly ratio in our training dataset. Additionally, for IF, we set the maximum sample size to 100 and use the default values for the rest.

We use the open-source implementation of CoMTE[7]. CoMTE requires an ML model that returns classification probabilities and the training dataset, whereas autoencoder-based models, including *Prodigy* predict an anomaly based on a threshold. To address this, we slightly modify the existing implementation of *BruteForceSearch* and *OptimizedSearch* classes.

6 EVALUATION

We compare the F1-score of *Prodigy* and baselines in the first section across two datasets. The F1-score is the harmonic mean of precision and recall, and we use the macro average F1-score, which treats all classes equally. This is crucial in imbalanced data sets, where healthy samples outnumber anomalous ones. For the rest of the paper, F1-score refers to the macro average F1-score. In the second section, we evaluate the performance of *Prodigy* with two production system experiments.

6.1 Controlled Experiments

We compare *Prodigy* with the baselines and report the average F1-scores based on the 5-fold cross-validation. Figure 5 shows the comparison results. We observe that, in the Eclipse dataset, LOF performs the worst among all the baselines, with an average F1-score of 0.15. Conversely, in the Volta dataset, Random Prediction lags behind with an F1-score of 0.39. Majority Label Prediction performs slightly better than Random Prediction, with an F1-score of approximately 0.47. IF performs significantly better on the Volta dataset, with an F1-score of 0.86, compared to the Eclipse dataset, which only achieves an F1-score of 0.31. The difference in performance is attributed to the disparity in the anomaly ratio in the test datasets. While the anomaly ratio in the training dataset is 10% for both datasets, the Eclipse dataset has significantly more anomalous samples (i.e., 90% anomaly ratio) due to the data collection strategy. As a result, setting the contamination ratio to 10% during the training of isolation forest leads to poor performance on the Eclipse dataset. USAD performs better on the Volta dataset with an F1-score of 0.84, while it performs worse on the Eclipse dataset with a 0.68 F1-score. *Prodigy* achieves a 0.95 and 0.88 F1-score on Eclipse and Volta datasets, respectively, and outperforms other baselines.

[7] https://github.com/peaclab/CoMTE

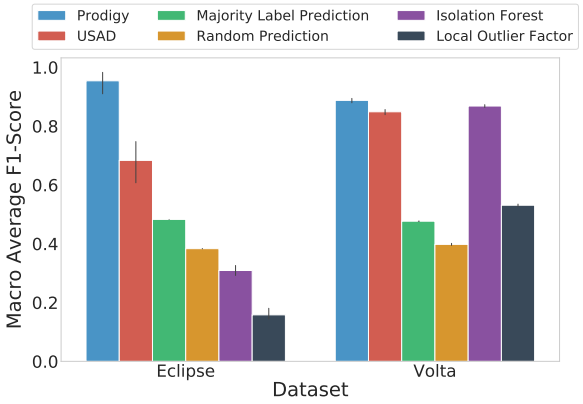

Figure 5: Comparison of *Prodigy* with baselines. *Prodigy* reaches 0.95 and 0.88 F1-scores in Eclipse and Volta, outperforming the baselines.

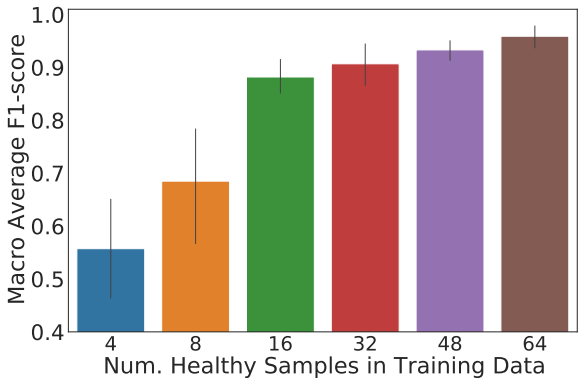

Figure 6: We run a separate experiment on Eclipse to demonstrate *Prodigy*'s ability to function effectively with a limited amount of production system data. The figure shows the change in F1-score in the test dataset with respect to different numbers of healthy samples in the training dataset.

6.2 Production System Experiments

In this section, we conduct two experiments on Eclipse. The first experiment aims to demonstrate that the proposed framework can work with a small amount of production system data and still achieve a satisfactory anomaly detection score. We select 4 previously compiled applications: LAMMPS, sw4, sw4Lite, and ExaMiniMD, to cover real and proxy applications. We run each application 5 times without injecting any anomalies on 4 compute nodes. We select the *memleak* anomaly to demonstrate in our experiment because it is one of the most common anomaly types. We also run each

application 5 times with the selected anomaly. The distribution of the collected dataset is as follows: 160 samples in total, 80 samples for anomalous and healthy labels. We experiment with different numbers of healthy samples existing in the training dataset and repeat the selection process 10 times in each case. The test dataset has all the anomalous and remaining healthy samples because we want to ensure our framework can work well with unseen healthy samples. Figure 6 shows the F1-score of *Prodigy* when the training dataset has different numbers of healthy samples. When the training dataset has only 4 samples (i.e., 1 job that runs on 4 compute nodes), *Prodigy* achieves a 0.58 F1-score. *Prodigy* achieves an average **F1-score of 0.9 using only 16 healthy samples**. When the number of healthy samples increases, approximately 60 samples, *Prodigy* achieves a 0.96 F1-score. This experiment shows that *Prodigy* can work with a small amount of production system data if a user wants to deploy to an existing system.

We also investigate the explanations generated by CoMTE to better understand the characteristics of anomalous samples and potential fixes. *MemFree::meminfo* and *pgrotated::vmstat* are the top two metrics CoMTE returned as an explanation for a sample predicted as anomalous. Figure 7 shows the predictions for each compute node in the chosen job ID and raw time series corresponding to the explanation metrics. Each plot shows all compute nodes in the chosen job ID. The *MemFree::meminfo* metric reports the amount of free and available memory. It has a clear decreasing trend for the anomalous compute nodes, whereas healthy compute nodes have a constant memory usage pattern for most of the application run. *pgrotated::vmstat* metric reports the number of pages that have been rotated to and from the swap space since the last boot of the system. When the value of *pgrotated::vmstat* metric is high, it may indicate that the system is experiencing memory pressure and is being forced to swap out pages to free up physical memory. We compare values of *pgrotated::vmstat* metric for healthy and anomalous compute nodes separately and observe that in some cases anomalous compute nodes have higher values, and in some cases, they are lower. It is possible that memory leakage does not result in swapping, i.e., when the leaked memory is still within the program's address space, it is not swapped out to disk. Even though values of *pgrotated::vmstat* metric do not exhibit a consistent trend, an application domain expert should have a much better starting point for debugging their application given this information.

The goal of the second experiment is to demonstrate that the *Prodigy* can detect performance anomalies in the wild, i.e., we do not inject synthetic anomalies. We collaborate with a plasma-physics domain expert who had identified that their runs of Empire [12] occasionally run with degraded performance. We run this application with the same input parameters multiple times to observe a variation on 4 compute nodes. We observe that 7 jobs are completed in around 60 minutes, labeled as healthy, and 2 jobs that take 10-30% longer, labeled as anomalous. HPC domain experts determine that the application occasionally has degraded I/O performance due to apparent backend Lustre file system issues. We train our framework with the available healthy jobs (28 samples in total) and test the performance with the anomalous jobs (8 samples). *Prodigy* detects anomalies in 7 samples out of 8. Even though we have 7 healthy jobs, *Prodigy* can extract the characteristics of healthy samples and

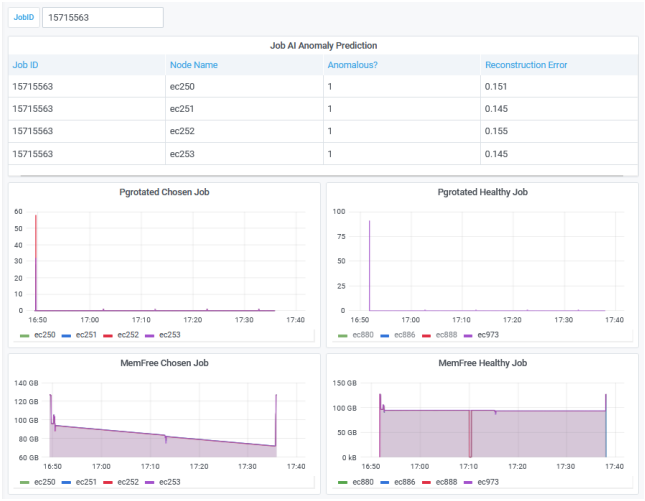

Figure 7: The anomaly detection results and CoMTE explanations for a job ID (shown as "Chosen Job") with the *memleak* anomaly from the Empire application runs. We compare the metrics provided by CoMTE with the healthy version (no synthetic anomalies injected) of the same application.

achieve an 88% accuracy for the samples labeled by subject matter experts.

As a last step of our production system experiments, we measure the average inference time to generate predictions for all the samples in Volta and Eclipse test datasets. To predict 18,947 and 14,589 samples takes 3.28 and 2.5 seconds on average for Eclipse and Volta test datasets, respectively. We average the results over ten runs using a compute node with two 14-core 2.4 GHz Intel Xeon E5-2680v4 processors.

7 CONCLUSION & FUTURE WORK

In this paper, we propose *Prodigy*, a VAE-based unsupervised framework that detects performance anomalies in compute nodes. The framework assumes access to healthy labeled samples during the training phase to learn the healthy application run characteristics. Our evaluation shows that the proposed framework achieves a 0.95 F1-score on a dataset collected from a production HPC system, Eclipse. Additionally, we apply a state-of-the-art counterfactual explainability framework, CoMTE, to the anomaly detection problem, which helps HPC administrators understand the root cause of anomalies. We demonstrate the practicality of our framework by deploying it on a production HPC system with 1488 compute nodes, providing job and node-level analysis through the deployed framework.

As part of future work, we plan to support a fully unsupervised pipeline for *Prodigy*. This direction is predicated on our assumption of exclusively healthy samples during the training phase, while the telemetry data from production systems may contain a small percentage of anomalous samples. Another potential avenue for future exploration is exploring heterogeneous computing systems with GPU and CPU compute nodes. Naturally, the telemetry data

generated by GPUs and CPUs differs in terms of metrics and granularity. Thus, there is a necessity for frameworks that can effectively operate within heterogeneous system environments.

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] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. 2014. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 154–165.

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Ann Gentile, Sophia Lefantzi, Steve Monk, Jeff Ogden, Mahesh Rajan, and Joel Stevenson. 2015. Toward rapid understanding of production HPC applications and systems. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, 464–473.

[3] Burak Aksar, Benjamin Schwaller, Omar Aaziz, Vitus J Leung, Jim Brandt, Manuel Egele, and Ayse K Coskun. 2021. E2EWatch: an end-to-end anomaly diagnosis framework for production HPC systems. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer, 70–85.

[4] Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J Leung, Jim Brandt, Brian Kulis, and Ayse K Coskun. 2022. ALBADross: Active Learning Based Anomaly Diagnosis for Production HPC Systems. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, 369–380.

[5] Burak Aksar, Yijia Zhang, Emre Ates, Benjamin Schwaller, Omar Aaziz, Vitus J Leung, Jim Brandt, Manuel Egele, and Ayse K Coskun. 2021. Proctor: A semi-supervised performance anomaly diagnosis framework for production hpc systems. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*. Springer, 195–214.

[6] Roy Assaf and Anika Schumann. 2019. Explainable Deep Neural Networks for Multivariate Time Series Predictions. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI*. International Joint Conferences on Artificial Intelligence Organization, Macao, China, 6488–6490.

[7] Emre Ates, Burak Aksar, Vitus J Leung, and Ayse K Coskun. 2021. Counterfactual explanations for multivariate time series. In *International Conference on Applied Artificial Intelligence (ICAPAI)*. IEEE, 1–8.

[8] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. 2019. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *Proceedings of the 48th International Conference on Parallel Processing*. ACM, 1–10.

[9] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A Zuluaga. 2020. Usad: Unsupervised anomaly detection on multivariate time series. In *Proceedings of the 26th SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, New York, NY, USA, 3395–3404.

[10] Dor Bank, Noam Koenigstein, and Raja Giryes. 2020. Autoencoders. *CoRR* abs/2003.05991 (2020). arXiv:2003.05991

[11] Elisabeth Baseman, Sean Blanchard, Nathan DeBardeleben, Amanda Bonnie, and Adam Morrow. 2016. Interpretable anomaly detection for monitoring of high performance computing systems. In *Outlier Definition, Detection, and Description on Demand Workshop at SIGKDD*. ACM, 1–27.

[12] Matthew Tyler Bettencourt, Sidney Shields, Kristian Beckwith, Keith Cartwright, Eric C Cyr, Richard Michael Jack Kramer, Paul Lin, William McDoniel, Sean Miller, Roger P. Pawlowski, Edward Geoffrey Phillips, and Nathan V. Roberts. 2019. EMPIRE: Sandia's Next Generation Plasma Tool ? Kokkosifying EMPIRE-Fluid. (4 2019).

[13] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, New York, NY, USA, 1–12.

[14] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. 2019. A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems. *Engineering Applications of Artificial Intelligence* 85 (2019), 634–644.

[15] Andrea Borghesi, Antonio Libri, Luca Benini, and Andrea Bartolini. 2019. Online anomaly detection in hpc systems. In *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 229–233.

[16] James M Brandt, David DeBonis, Ann C Gentile, Jim Lujan, Cindy Martin, David J Martinez, Stephen Lecler Olivier, Kevin Pedretti, Narate Taerat, and Ron Velarde. 2015. *Enabling Advanced Operational Analysis Through Multi-subsystem Data Integration on Trinity*. Technical Report. Sandia National Lab. (SNL-CA), Livermore, CA (United States); Sandia National .

[17] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W Kempa-Liehr. 2018. Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package). *Neurocomputing* 307 (2018), 72–77.

[18] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run variability on Xeon Phi based Cray XC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, New York, NY, USA, 1–13.

[19] Bruno L Dalmazo, Joao P Vilela, Paulo Simoes, and Marilia Curado. 2016. Expedite feature extraction for enhanced cloud anomaly detection. In *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1215–1220.

[20] Ailin Deng and Bryan Hooi. 2021. Graph Neural Network-Based Anomaly Detection in Multivariate Time Series. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 5 (May 2021), 4027–4035.

[21] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. 2014. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *International parallel and distributed processing symposium*. IEEE, 155–164.

[22] Alan H Gee, Diego Garcia-Olano, Joydeep Ghosh, and David Paydarfar. 2019. Explaining deep classification of time-series data with learned prototypes. 2429 (2019), 15.

[23] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)* 28, 1 (1979), 100–108.

[24] Theodore P. Hill. 1995. A Statistical Derivation of the Significant-Digit Law. *Statist. Sci.* 10, 4 (1995), 354 – 363.

[25] Tsung-Yu Hsieh, Suhang Wang, Yiwei Sun, and Vasant Honavar. 2021. Explainable Multivariate Time Series Classification: A Deep Neural Network Which Learns to Attend to Important Variables As Well As Time Intervals. In *Proceedings of the 14th International Conference on Web Search and Data Mining (WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 607–615.

[26] Jannis Klinkenberg, Christian Terboven, Stefan Lankes, and Matthias S Müller. 2017. Data mining-based analysis of HPC center operations. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 766–773.

[27] Vitus Joseph Leung, Michael A Bender, David P Bunde, and Cynthia Ann Phillips. 2003. *Algorithmic support for commodity-based parallel computing systems*. Technical Report. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA .

[28] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NeurIPS 30*. 4765–4774.

[29] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. 2017. An empirical survey of performance and energy efficiency variation on intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*. ACM, 1–8.

[30] Matthew L Massie, Brent N Chun, and David E Culler. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[31] Martin Molan, Andrea Borghesi, Daniele Cesarini, Luca Benini, and Andrea Bartolini. 2023. RUAD: Unsupervised anomaly detection in HPC systems. *Future Generation Computer Systems* 141 (2023), 542–554.

[32] Alessio Netti, Michael Ott, Carla Guillen, Daniele Tafani, and Martin Schulz. 2022. Operational data analytics in practice: experiences from design to deployment in production HPC environments. *Parallel Comput.* 113 (2022), 102950.

[33] Alessio Netti, Woong Shin, Michael Ott, Torsten Wilde, and Natalie Bates. 2021. A conceptual framework for HPC operational data analytics. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, 596–603.

[34] Gence Ozer, Alessio Netti, Daniele Tafani, and Martin Schulz. 2020. Characterizing HPC Performance Variation with Monitoring and Unsupervised Learning. In *High Performance Computing: ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21–25, 2020, Revised Selected Papers 35*. Springer, 280–292.

[35] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London,*

Edinburgh, and Dublin Philosophical Magazine and Journal of Science 50, 302 (july 1900), 157–175.

[36] Danny Pfeffermann and J Allon. 1989. Multivariate exponential smoothing: Method and practice. *International Journal of Forecasting* 5, 1 (1989), 83–98.

[37] Gregory C Reinsel. 2003. *Elements of multivariate time series analysis*. Springer Science & Business Media.

[38] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. 1135–1144.

[39] Stephen J Roberts, Dirk Husmeier, Iead Rezek, and William Penny. 1998. Bayesian approaches to Gaussian mixture modeling. *Transactions on Pattern Analysis and Machine Intelligence* 20, 11 (1998), 1133–1142.

[40] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[41] U. Schlegel, H. Arnout, M. El-Assady, D. Oelke, and D. A. Keim. 2019. Towards A Rigorous Evaluation Of XAI Methods On Time Series. In *IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. 4197–4201.

[42] Thomas Schreiber and Andreas Schmitz. 1997. Discrimination power of measures for nonlinearity in a time series. *Physical Review E* 55, 5 (1997), 5443.

[43] Benjamin Schwaller, Nick Tucker, Tom Tucker, Benjamin Allan, and Jim Brandt. 2020. HPC system data pipeline to enable meaningful insights through analysis-driven visualizations. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 433–441.

[44] Burr Settles. 2009. Active learning literature survey. (2009).

[45] David Skinner and William Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *Proceedings of the Workload Characterization Symposium*. IEEE, 137–149.

[46] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. 2014. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications* 28, 2 (2014), 129–173.

[47] N Tajunisha and V Saravanan. 2010. An increased performance of clustering high dimensional data using Principal Component Analysis. In *First International Conference on Integrated Intelligent Computing*. IEEE, 17–21.

[48] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. 2022. TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1201–1214.

[49] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. 2018. Online diagnosis of performance variation in HPC systems using machine learning. *Transactions on Parallel and Distributed Systems* 30, 4 (2018), 883–896.

[50] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation* 31, 7 (2019), 1235–1270.

[51] Yijia Zhang, Taylor Groves, Brandon Cook, Nicholas J Wright, and Ayse K Coskun. 2020. Quantifying the impact of network congestion on application performance and network metrics. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, 162–168.