Effective Node-Level Anomaly Detection in HPC Systems via Coarse-Grained Clustering and Fine-Grained Model Sharing

Sibo Xia
Nankai University
Tianjin, China
xiath@mail.nankai.edu.cn

Yongqian Sun*
Nankai University
Tianjin, China
Tianjin Key Laboratory of Software Experience and Human Computer Interaction
Tianjin, China
sunyongqian@nankai.edu.cn

Xijie Pan
Nankai University
Tianjin, China
panxijie@mail.nankai.edu.cn

Yuan Yuan*
National University of Defense Technology
Changsha, China
yuanyuan@nudt.edu.cn

Shenglin Zhang
Nankai University
Tianjin, China
Haihe Laboratory of Information Technology Application Innovation
Tianjin, China
zhangsl@nankai.edu.cn

Shaoyu Hu
Nankai University
Tianjin, China
shaoyu.hu@mail.nankai.edu.cn

Lei Tao
Nankai University
Tianjin, China
leitao@mail.nankai.edu.cn

Yuqi Li
National University of Defense Technology
Changsha, China
National Supercomputer Center in Tianjin
Tianjin, China
liyq@nscc-tj.cn

Jinghua Feng
National Supercomputer Center in Tianjin
Tianjin, China
fengjh@nscc-tj.cn

Abstract

High-performance computing (HPC) systems are crucial for scientific advancement and engineering breakthroughs. Unexpected performance degradation or system failures can severely impact these endeavors. This paper introduces *NodeSentry*, a novel unsupervised anomaly detection framework tailored for compute nodes of large-scale HPC systems. *NodeSentry* leverages a combined approach of coarse-grained clustering and fine-grained model sharing to effectively address the challenges posed by the massive node scales, frequent job transitions, and complex patterns characteristic of modern HPC deployments. Evaluation on two real-world HPC datasets demonstrates *NodeSentry*'s superior performance, achieving an F1-score exceeding 0.876. This represents a 0.560 average improvement over existing best baseline methods, while simultaneously reducing training overhead by an average of 45.69%. Furthermore, to promote reproducibility and contribute to the broader research community, we open-source *NodeSentry*'s codebase and introduce a novel clustering adjustment and anomaly labeling tool specifically designed for HPC systems.

CCS Concepts

• **General and reference → Performance**.

Keywords

High Performance Computing, Anomaly Detection, Clustering, Model Sharing

ACM Reference Format:
Sibo Xia, Yongqian Sun, Xijie Pan, Yuan Yuan, Shenglin Zhang, Shaoyu Hu, Lei Tao, Yuqi Li, and Jinghua Feng. 2025. Effective Node-Level Anomaly Detection in HPC Systems via Coarse-Grained Clustering and Fine-Grained Model Sharing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759794

*Yongqian Sun and Yuan Yuan are the corresponding authors.
TKL-SEHCI, HL-IT, and NSCC-TJ are short for Tianjin Key Laboratory of Software Experience and Human Computer Interaction, Haihe Laboratory of Information Technology Application Innovation, and National Supercomputer Center in Tianjin, respectively.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759794

Table 1: A list of common anomalies in HPC systems.

Level	Type
CPU	CPU Overload, Cache Failure, *etc.*
Memory	Memory Exhaustion, Memory Leak, *etc.*
Disk	Silent Data Corruption, Disk Full, *etc.*
Network	Network Partition Bugs, Network Congestion, *etc.*
Kernel/OS	Resource Contention, Page Allocation Error, *etc.*

1 Introduction

High-performance computing (HPC) systems play a crucial role in various data-intensive applications in social and scientific domains, such as weather forecasting, special effects rendering, and aerospace [4]. These systems typically comprise a substantial number of **compute nodes** interconnected by high-bandwidth, low-latency networks, forming a cluster or supercomputer [8]. Each node (we use "**node**" as a shorthand for "compute node" in this paper) is equipped with multiple processor cores and significant memory capacity [24], and numerous nodes often pool their resources to tackle a single, large-scale computational **job**. The intricate and expansive architecture and highly dynamic job scheduling, significantly amplify the likelihood of system malfunctions [4, 14, 50], *e.g.*, the world's top-ranked supercomputer, Frontier, initially experienced a mean time between failures of just a few hours [44]. Consequently, the real-time identification of performance anomalies, particularly before they escalate into system failures, is of paramount importance. The details of anomalies for nodes in HPC systems are provided in Tab. 1 [11, 28, 33]. Common remediation steps following detection include node isolation, task restarts, and detailed analysis by operators. A prevalent way to achieve this is to monitor nodes and detect anomalies in a real-time manner [34].

To detect the anomalies of nodes, operators closely monitor and collect large-scale performance metrics from these nodes, *e.g.*, core utilization rates and memory usage percentages [20]. These metrics are aggregated in the form of multivariate time series (MTS) at predefined time intervals. Fig. 1 illustrates a subset of these metrics from three nodes. We observed fundamental differences between the MTS of HPC systems and those observed in other application scenarios. Specifically, when compared with cloud data centers [18, 38], microservice systems [39, 49], and cellular base stations [27, 39], HPC systems exhibit three distinctive MTS characteristics:

1) **High node scale and metric dimension:** HPC systems typically consist of tens of thousands of nodes. Each node generates a vast number of metrics, *i.e.*, 3,014 metrics in our scenario (only a few are shown in Fig. 1), due to a large number of cores, memory, network interfaces, and other components per node. In contrast, the MTS for a node comprises only dozens of metrics in a typical data center [18, 38].
2) **Dynamic job transition and job pattern correlation:** As is well known, jobs in HPC are typically complex and require distributed collaborative execution across multiple nodes, *e.g.*, (a) in Node-1 and (f) in Node-3. While from the perspective of a single node, jobs are constantly being switched, *e.g.*, (f), (g), (h), and (i) in Node-3 are all different jobs, where (h) represents an idle waiting state, which can be regarded as a special type of job. It can be observed that nodes executing the same job usually have similar patterns, *i.e.*, (a) and (f). However, different jobs may also exhibit similar patterns, such as (e) and (i). Moreover, this correlation implies that the patterns of jobs on other nodes can provide valuable insights for understanding the behavior of jobs on the target node, even in the absence of repeated job information.
3) **Variation between sub-patterns within the same job:** Although different MTS segments may exhibit similar patterns, the fine-grained patterns within a single segment (corresponding to a single job) are not static. For example, the last part and the preceding parts of segments (e) and (i) show significant differences, which we refer to as sub-pattern 1 and sub-pattern 2, respectively. This is because the specific tasks within a continuous job segment can vary over time.

Automatic anomaly detection in HPC systems has garnered significant interest over the years [3–5, 10, 30, 32, 37, 42]. However, existing methods, which assume MTS with stable patterns and strong periodicity, fall short of effectively addressing anomaly detection for nodes exhibiting these characteristics. Regarding Characteristic 1, some methods [10, 22, 30] train a model for each node, with the training cost increasing exponentially as the node scale and metric dimension grows. Concerning Characteristic 2, the inability to accumulate stable, long-term continuous pattern data leads to significantly poor training and detection performance for some methods [5, 10, 30, 32, 42]. Although some methods [3, 4, 37] have considered these characteristics, they have not accounted for the impact of Characteristic 3, failing to effectively model the variations between sub-patterns.

Fortunately, in our scenario, we can easily obtain every job's start times, end times, and execution nodes from the management system using Slurm's (a job scheduling tool widely adopted by HPC systems) `sacct` command [1]. By using the start and end times of jobs, we can segment the continuous time series collected on the node into multiple segments, each representing the time series pattern of a specific job (will be elaborated in § 3.2). As a result, we can abandon the approach of training a model for each node or each job individually. Instead, we adopt the following strategy: First, we segment all nodes into MTS segments based on jobs. Then, we cluster these segments. After that, we train a model for each cluster, which is capable of simultaneously focusing on various sub-patterns. When a new job is to be detected, only a limited amount of data is required to match the most appropriate patterns, enabling effective anomaly detection through the corresponding model. However, implementing this targeted strategy for anomaly detection presents two major challenges (will be elaborated in § 2.1).

1) **Clustering the segments of different lengths for coarse-grained patterns:** Clustering thousands of dimensions of MTS is impractical due to the immense computational complexity, necessitating dimensionality reduction. On the other hand, job-based segmentation yields segments of different lengths, the clustering of which poses a primary challenge. Furthermore, adaptive clustering methods that account for different lengths typically incur high computational complexity [26, 49]. These limitations hinder the effective execution of pattern clustering by existing methods.

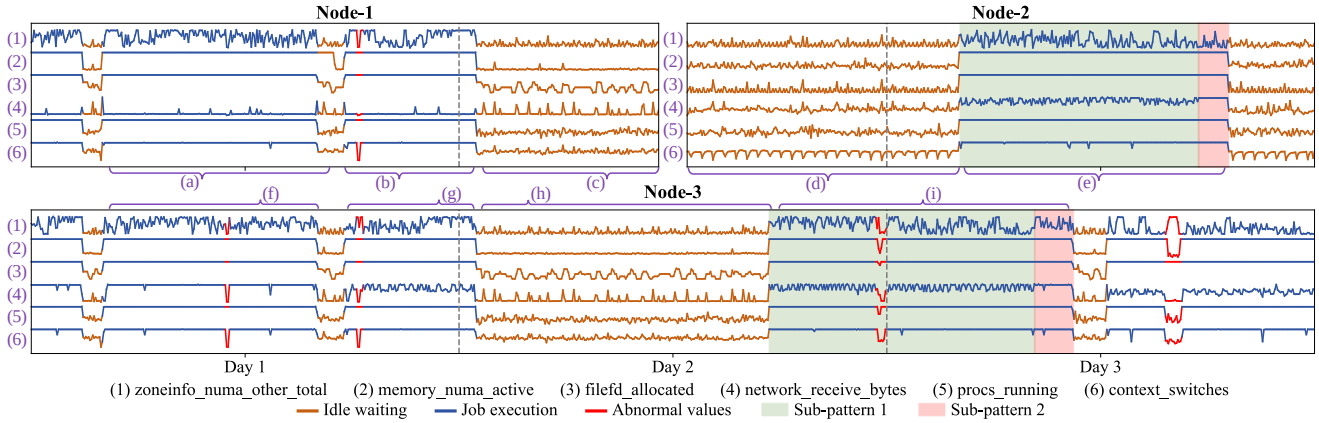

Figure 1: The MTS of three nodes shows 6-metric examples with the duration of 1.5 days for two nodes and 3 days for one node. similar pattern pairs: (a)–(f), (c)–(h), and (e)–(i); different pattern pairs: (b)–(g) and (d)–(h).

2) **Identification and model sharing for fine-grained sub-patterns:** While clustering methods can successfully group segments that exhibit similar global trends, the segments within the same cluster might still contain intricate sub-patterns that remain insufficiently identified. Another significant challenge lies in effectively recognizing and accommodating these fine-grained sub-patterns to train a shared model that ensures accurate anomaly detection and model generalization.

To tackle the aforementioned challenges, we propose *NodeSentry*, an unsupervised anomaly detection framework for nodes in HPC systems. *NodeSentry* first extracts comprehensive features and utilizes Hierarchical Agglomerative Clustering (HAC) [31, 49] to perform coarse-grained pattern clustering, aiming to quickly classify and identify major trends. *NodeSentry* then employs a combination of Transformer and Mixture of Experts (MoE) [17, 36] to achieve model sharing and optimization at a more fine-grained level, which allows for handling sub-patterns within clusters and enhances the model's generalization and detection capabilities. The main contributions of this paper are summarized as follows:

1) For the first time, we summarize the unique characteristics of compute nodes in HPC systems. Inspired by them, we propose a novel anomaly detection framework via coarse-grained clustering and fine-grained model sharing, *NodeSentry*, which significantly enhances the reliability and stability of HPC systems.
2) We extract comprehensive statistical, temporal, and spectral features and utilize HAC to represent the segments of different lengths as fixed-width vectors for pattern clustering, addressing Challenge 1. Subsequently, we integrate both the self-attention mechanism of the Transformer and the model-sharing advantages of MoE to enhance the detection performance and generalization capabilities, addressing Challenge 2.
3) We conduct extensive experiments on two distinct datasets collected from production systems. The results show that *NodeSentry* achieves the average F1-score of 0.876 and 0.891, respectively, outperforming the best baseline methods by 0.562 and 0.558, while reducing training overhead by 77.93% and 13.45% compared to the fastest deep learning baseline methods, respectively.
4) To ensure better reproducibility, we have made our sample dataset, source code, clustering adjustment tool and anomaly labeling tool for MTS in HPC systems publicly available [1].

2 Challenges and Techniques

In this section, we discuss the challenges in detail and introduce some techniques that will be utilized to tackle these challenges. Additionally, we define the specific problem of anomaly detection.

2.1 Challenges

Challenge 1: Clustering the segments of different lengths for coarse-grained patterns.

As shown in Fig. 1, there are different jobs, which are recorded in the job management system. However, directly training individual models for each job is impractical due to insufficient training data and high computational costs. Therefore, we need to cluster similar jobs together to train shared models, which requires segmenting the data according to the different jobs within each node and effectively clustering these MTS segments, making it a difficult problem.

The high metric dimensionality makes it infeasible to directly perform clustering on the raw MTS. Firstly, the sheer volume of dimensions leads to a curse of dimensionality. As the number of dimensions increases, the contrast between the distances of different data points diminishes, making it difficult to distinguish between clusters. Additionally, the computational complexity of clustering algorithms escalates exponentially with the number of dimensions, which may not be practical for large-scale nodes.

The varying job durations in HPC systems result in MTS segments of unequal lengths, which poses significant difficulties for clustering. Existing clustering methods for segments of different lengths can be broadly categorized into two types: shape-based methods and deep learning-based methods. Dynamic Time Warping (DTW) [26] is the most typical shape-based clustering method, which is computationally intensive and inefficient. Using this method to cluster a week's worth of data would take 3.8 months, which is unacceptable for practical applications. On the other hand, deep learning-based

[1] https://github.com/AIOps-Lab-NKU/NodeSentry/

methods require extensive amounts of data to learn effective representations [38, 46–49]. In HPC systems, however, jobs can be very short-lived, and the limited length of MTS segments may not provide sufficient information to support the training of deep learning models.

Over above, reducing the dimensionality and extracting statistical, temporal, and spectral features of metrics for clustering becomes an appropriate approach [9, 15, 25, 37]. Dimensionality reduction methods, help mitigate the curse of dimensionality by transforming the data into a lower-dimensional space while preserving important information. On the other hand, the feature extraction method aims to identify the most relevant and discriminative features by analyzing the statistical, temporal, and spectral characteristics of the data. By extracting meaningful features and compressing the time dimension, we can improve clustering efficiency and achieve better performance.

Challenge 2: Identification and model sharing for fine-grained sub-patterns.

From Characteristic 3, it is evident that in HPC systems, MTS can exhibit distinct sub-patterns even within the same job. Following the clustering of similar patterns into distinct clusters, the subsequent challenge lies in training a shared model for each cluster while effectively addressing the inherent diversity of sub-patterns.

The traditional sharing strategies, which include using cluster centroids for model training [49] *and basing data sharing on metrics dimensionality* [18]*, each have limitations.* Specifically, using cluster centroids for model training [49] involves grouping similar patterns into clusters and training a model based on the centroid of each cluster. This approach assumes homogeneity within clusters, which can undermine the model's ability to generalize effectively to complex sub-patterns. Meanwhile, data sharing based on metric dimensionality [18] captures cross-metric correlations in MTS, treating each metric as a position to handle diverse data across patterns, which disregards temporal dependencies and compromises the model's capacity to adapt to the nuanced variations of sub-patterns. These strategies often struggle with insufficient representativeness of data, leading to a curtailment in the model's generalization capacity.

In contrast, integrating ensemble learning into the sharing strategy offers a novel perspective [12, 54]. By leveraging the strengths of multiple individual learners, ensemble learning can effectively address the inherent diversity of sub-patterns within clusters. This approach not only enhances the model's ability to generalize across similar patterns but also improves its adaptability to diverse sub-patterns. Specifically, ensemble methods can aggregate results from individual learners trained on different subsets of data within the same cluster, thereby providing a more comprehensive and robust representation of the underlying patterns. This strategy thus provides a more effective solution for training shared models in HPC systems. MoE is an excellent method for implementing this strategy [23], and Transformer is well-suited for integrating experts who focus on different subtle sub-patterns [16, 21].

2.2 Techniques

Through rigorous analysis, we identify a set of targeted techniques that address the core issues effectively. This section delves into the techniques we've selected, outlining theoretical foundations.

Hierarchical Agglomerative Clustering. HAC is a bottom-up dendrogram-based clustering technique that offers flexibility in selection strategies and provides a clear interpretation of the clustering structure. HAC allows for multi-level clustering results, which is beneficial for discovering patterns. HAC can handle clusters of varying sizes and shapes, unlike centroid-based clustering algorithms (*e.g.*, k-means) that assume circular or spherical clusters.

Transformer Neural Networks. Transformer utilizes the self-attention mechanism to achieve global context modeling, enabling better capturing of long-range dependencies in MTS [40, 53]. Its parallel computing capability provides higher speed and scalability during both training and inference processes. Additionally, the inclusion of a feed-forward network (FFN) contributes to improved training stability and expressive power of the model. These characteristics have made the Transformer the mainstream model in current natural language processing and MTS modeling tasks.

Mixture of Experts. MoE combines the strengths of multiple experts and diverse knowledge representations, offering rich data features and strong pattern-capturing abilities [45, 51]. Compared to dense models, MoE can enjoy its benefits while maintaining relatively lower computational costs [21]. Furthermore, these characteristics can also facilitate model sharing, reducing the training data requirements for each expert model and allowing the model to learn more knowledge from a limited time range of MTS. The combination of MoE within the Transformer has gained extensive attention and application in model training tasks.

2.3 Problem Definition

Operators need to analyze the real-time collected MTS. Consider a large-scale HPC system with $\mathcal{N}$ nodes, where collecting MTS $\mathcal{M}$ is associated with each node. In a $\mathcal{T}$-length timestamped sequence of observations, $\mathcal{X} \in \mathbb{R}^{\mathcal{N}\times\mathcal{M}\times\mathcal{T}}$ represents MTS of all nodes at consecutive timestamps. For each node $n \in \{1, ..., \mathcal{N}\}$, the information recorded in the job scheduling list can be processed as $\mathcal{S}_n = \left\{(start_{n,j}, end_{n,j})\right\}_{j=1}^{\mathcal{J}_n}$, where $\mathcal{J}_n$ is the total number of jobs, $start_{n,j}$ and $end_{n,j}$ are the start and end timestamp of the j-th job.

Based on the above definition, the problem of anomaly detection is formulated as follows: Given a training input $\mathcal{X}$, for $\hat{\mathcal{T}}$-length $\hat{\mathcal{X}}$ to be tested, we need to determine $\mathcal{Y} \in \{0, 1\}^{\mathcal{N}\times\mathcal{T}}$, where $\mathcal{Y}$ to present whether each node at the timestamps of the testing is anomalous (1 denotes an anomaly).

3 Methodology

3.1 Design Overview

The overarching structure of the framework is depicted in Fig. 2, which encompasses two main phases: the offline model training phase and the online anomaly detection phase.

During the offline model training phase, *NodeSentry* first preprocesses the raw MTS to eliminate differences in format, range, and patterns across various nodes. Subsequently, to identify the patterns of the jobs and reduce model training overhead, *NodeSentry* performs coarse-grained clustering on the processed MTS segments. This approach enables the model to train on the clusters rather than on individual MTS segments. To effectively recognize and adapt to the sub-patterns within the same cluster, *NodeSentry* assigns expert

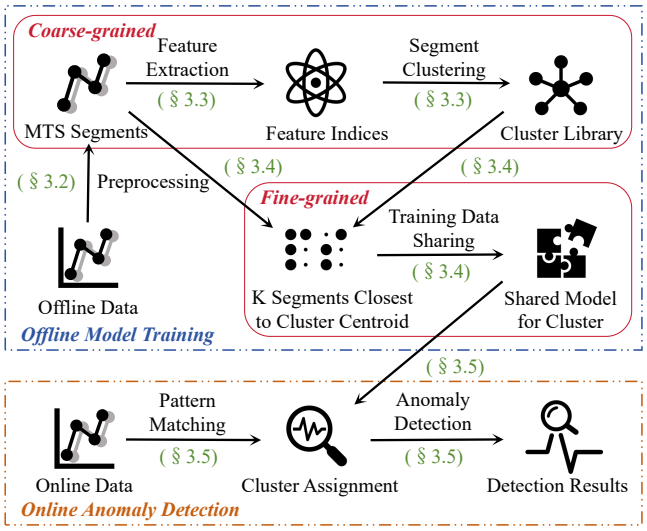

Figure 2: The overall framework of *NodeSentry*.

networks to different segments within the cluster for model sharing. Each expert network can focus on processing specific sub-patterns, thereby enhancing performance and generalization capabilities. In the online anomaly detection phase, *NodeSentry* extracts features from preprocessed MTS to match patterns in the cluster library. Subsequently, the appropriate model is dynamically assigned to identify anomalies in the target nodes.

3.2 MTS Preprocessing

For the following three reasons, we need to conduct preprocessing operations: 1) To ensure clean and representative data, we apply linear interpolation to missing values. 2) To reduce complexity while preserving critical information, we perform node-level aggregation of core metrics, yielding stable node-status insights. 3) To capture meaningful patterns and mitigate continuous-metric ambiguity, we implement job-based segmentation at transition points, isolating node behaviors for specific job analysis. These constitute the core components of our standardized four-step preprocessing pipeline:

Cleaning: In real-world production environments, data missing may occur during the data collection and transmission process [43]. When values are missing, we fill in these positions linearly using the nearby observed values.

Reduction: As highlighted in Challenge 1, dimensionality reduction is essential due to the high dimensionality of MTS. We employ a combined approach utilizing two techniques to achieve significant reduction while minimizing information loss: dimensionality reduction based on metric semantics and similarity analysis.

(1) Dimensionality Reduction Based on Metric Semantics: To reduce data complexity and preserve essential node-state information, we aggregate similar metrics at the node level, combining only semantically identical metrics to maintain data integrity. This approach involves aggregating metrics with the same physical meaning (*e.g.*, CPU usage, memory consumption) [4, 37].

(2) Dimensionality Reduction Based on Similarity Analysis: In this step, we calculate the Pearson correlation coefficient between each pair of node-level metrics using the following formula:

$$r = \frac{\sum(\mathcal{X}_i - \bar{\mathcal{X}})(\mathcal{X}'_i - \bar{\mathcal{X}}')}{\sqrt{\sum(\mathcal{X}_i - \bar{\mathcal{X}})^2 \sum(\mathcal{X}'_i - \bar{\mathcal{X}}')^2}} \tag{1}$$

$\mathcal{X}$ and $\mathcal{X}'$ are two metrics for which the correlation coefficient needs to be calculated. For pairs of metrics with $r \geq 0.99$, these pairs typically exhibit nearly identical metric curves and tend to experience similar anomalies. We eliminate extremely similar metrics and retain only one, thereby without sacrificing the key patterns necessary for accurate anomaly detection.

After aggregating the per-core metrics and filtering out highly similar redundant indicators, we end up with several dimensions that are about a tenth of the original.

Standardization: Different metrics have varying units and ranges, which can result in varying contributions during anomaly detection. To deal with dimensionality differences between metrics with different units and ranges, we apply data standardization. When computing the mean $\mu_{i,j}$ and standard deviation $\sigma_{i,j}$ of the training data for each node-metric pair, we exclude the top and bottom 5% of extreme outliers for each metric to avoid skewing the data distribution [35]. These extreme outliers are defined as data points that deviate significantly from the normal data distribution, often due to measurement errors, data entry errors, or natural variation.

$$\mathcal{X}'_{i,j,t} = \frac{\mathcal{X}_{i,j,t} - \mu_{i,j}}{\sigma_{i,j}} \tag{2}$$

When standardizing the data, any remaining values that fall outside the range of -5 to +5 are clipped to these bounds. This ensures that the standardized data remains within a reasonable range, preventing undue influence from residual outliers.

Segmentation: Due to the different patterns exhibited by nodes during jobs, directly analyzing continuous MTS can introduce obscure key information. We employ a job-based segmentation approach to divide the MTS. Specifically, we first identify the job transition points (start and end times of jobs obtained from Slurm [1]). Then, we treat MTS between each job transition point as an independent segment, representing the node's continuous pattern in a specific job.

3.3 Coarse-grained Clustering

In HPC systems, the computational demands and durations of different jobs exhibit a high degree of uncertainty, leading to dynamic and frequent job transitions in the nodes. To simplify the problem space for anomaly detection in HPC systems and reduce the complexity of model training, we perform coarse-grained clustering of the processed MTS segments at a macro level. This approach helps capture representative patterns by grouping similar segments together, allowing the model to focus on these key clusters during training. The coarse-grained clustering process involves feature extraction and segment clustering to effectively reduce data complexity and enhance training efficiency.

Feature Extraction: Feature extraction is a crucial step in obtaining key information during data analysis. The process typically involves transforming MTS into a set of descriptive parameters that characterize the essence of the signal. Specifically, we use the Time Series Feature Extraction Library (TSFEL) [6] to extract 134

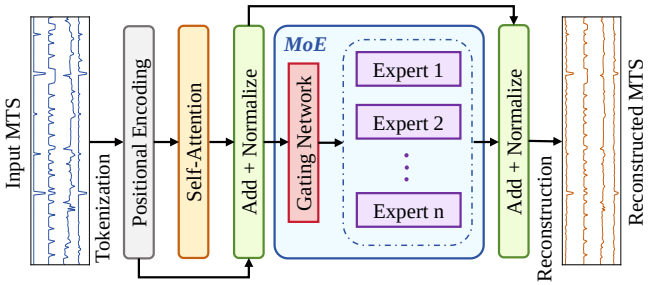

Figure 3: The overall architecture of model sharing.

interpretable feature indices for each metric. These features cover statistical, temporal, and spectral domains, including but not limited to median, absolute energy, and maximum power spectrum.

Segment Clustering: *NodeSentry* employs HAC and Euclidean distance for segment clustering. It is worth noting that operators do not require iterative attempts to determine the optimal number of clusters. We use the silhouette coefficient to measure clustering performance, which combines cohesion and separation [2]. A higher silhouette coefficient indicates greater distances between clusters and smaller distances within clusters, achieving high intra-cluster cohesion and low inter-cluster coupling.

3.4 Fine-grained Model Sharing

After grouping segments that exhibit similar patterns through coarse-grained clustering, segments within the same cluster may still contain complex sub-patterns that have not been fully identified. To capture fine-grained complex sub-patterns, *NodeSentry* utilizes the Transformer to enhance temporal context information without the need for fixed-size window inputs, thus improving detection performance [29, 41]. Furthermore, *NodeSentry* combines MoE to achieve model sharing by assigning expert networks to different segments within the clusters. Each expert network can focus on processing specific sub-patterns, thereby enhancing the performance and generalization capabilities [52].

The overall architecture for modeling the sub-patterns is illustrated in Fig. 3. The input MTS is tokenized, with each token being a vector composed of the metric values at each time point. These tokens are then processed through positional encoding. *NodeSentry* replaces the dense FFN layer in the Transformer with a sparse MoE layer. This layer consists of N FFNs denoted as experts and operates independently on the tokens. The MoE layer receives token x as input and routes it through the gated network to a set of N experts $\{E_i(x)\}_{i=1}^{N}$, and the routing variable W_r computes the result $h(x) = W_r \cdot x$. The gate value p_i of the expert i is calculated from $h(x)_i$ normalized by the softmax of the N experts of the layer and represents the relevance of each expert for this input. The top-k experts with the highest gate values, which are considered to be the most suitable for processing this data, will form the expert set n and the weighted results of these experts will determine the output of the MoE layer [16]:

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_j^N e^{h(x)_j}} \tag{3}$$

$$y = \sum_{i \in n} p_i(x) E_i(x) \tag{4}$$

Each token is allocated to a corresponding expert, whose results y will be fed into the decoder for reconstructing the data. During training, the routing variable W_r in the MoE layer is updated according to the experts' losses. The gated network learns how to select experts more efficiently for different data sub-patterns while the experts focus on learning for specific sub-patterns. The model is trained by minimizing the difference between the input data and the reconstructed data, which is calculated by the Weighted Mean Squared Error (WMSE):

$$WMSE = \frac{1}{\mathcal{M}} \sum_{i=1}^{\mathcal{M}} \mathcal{W}_i \left(\mathcal{X}_i - \mathcal{X}_i'\right)^2 \tag{5}$$

where $\mathcal{M}$ is the number of metrics, $\mathcal{W}_i$ is the weight, and $\mathcal{X}_i$ and $\mathcal{X}_i'$ represent the original and reconstructed data of the i-th metric of the node to be detected, respectively. To measure data stability, we employ the Mean Absolute Change (MAC):

$$MAC = \frac{1}{\mathcal{T}-1} \sum_{t=1}^{\mathcal{T}-1} |x_{t+1} - x_t| \tag{6}$$

where $\mathcal{T}$ is the number of data points, and x_t is the t-th observed values. We calculate the MAC for each metric based on the training data of each cluster to obtain $\mathcal{W}$.

Notably, utilizing K segments closest to the cluster centroid for training the shared model constitutes a form of data augmentation. As a result, we enhance the positional encoding in the Transformer to incorporate positional information within and between different segments. With this design, *NodeSentry* can naturally train a shared model for each cluster without requiring extensive training data and high training overhead.

3.5 Online Anomaly Detection

After completing the offline model training stage, we save the shared model for each cluster. During runtime, *NodeSentry* performs the same preprocessing steps in the offline phase. Then, *NodeSentry* utilizes the MTS for a short period (*e.g.*, 1 hour) following the node's job transition to extract features. *NodeSentry* calculates the distances between these features and the existing cluster centroids to match the most similar pattern. Finally, *NodeSentry* employs the corresponding shared model for detection.

The MTS from each node is fed into the shared model, which generates reconstructed data. The reconstruction error between the input and reconstructed data indicates the probability of the input MTS approaching normal behavior, also known as the anomaly score. In threshold selection, we dynamically set the threshold for the anomaly score. Specifically, we define a sliding window along the time axis. When the anomaly score exceeds the upper bounds of k-sigma, the data point is considered an anomaly [7]. In practice, operators often set a 3-sigma threshold.

Given the dynamic nature of data and the impossibility of capturing all possible patterns in advance, *NodeSentry* is driven by the need to leverage existing data to detect unseen patterns through the identification of the most similar ones. Specifically, when new patterns can be matched to any existing cluster within the cluster library, we conduct incremental fine-tuning of the existing model to adapt it to the changes in the new patterns. For patterns that cannot be matched, we perform clustering on these new patterns

Table 2: Detailed information of datasets.

Dataset	#Node	#Job	#Metric	Total Points	Anomaly Ratio
D1	1,294	13,379	3,014	106,850,650	0.16%
D2	30	1,430	773	1,555,200	0.04%

Table 3: An overview of monitoring metrics.

Category	Example	Number
CPU	cpu_seconds_total, perf_cpu_migrations_total, *etc.*	1378
Memory	numa_foreign_total, memory_kernel_stack_bytes, *etc.*	945
Filesystem	filefd_allocated, filesystem_files_free, *etc.*	254
Network	network_receive_bytes_total, sockstat_sockets_used, *etc.*	381
Process	processes_state, procs_running, procs_blocked, *etc.*	12
System	system_uptime, timex_status, ksmd_run, *etc.*	44

and train a new model accordingly. This strategy not only significantly enhances the efficiency of incremental and transfer learning compared to retraining models with large amounts of data but also effectively addresses the issue of frequent job transitions in nodes.

4 Evaluation

In this section, we evaluate the performance of *NodeSentry*, aiming to answer the following research questions:

- **RQ1:** Can *NodeSentry* achieve excellent anomaly detection performance through the two-stage strategy?
- **RQ2:** Does each module of *NodeSentry* contribute significantly to its performance?
- **RQ3:** What is the impact of *NodeSentry* after incremental training with a limited dataset?
- **RQ4:** How do the different hyperparameters affect *NodeSentry*?

4.1 Experimental Setup

4.1.1 Dataset. To address RQ1-4, we conduct extensive experiments on two datasets, D1 and D2, collected from the production environment of *NG-Tianhe*. We believe that the evaluation of a large-scale supercomputer system is sufficiently representative. They originate from arrays characterized by diverse node hardware designs.

To prevent data leakage, we partition the data into distinct training and test sets by considering their respective start times. Specifically, we employ data from the initial 60% of the time as the training set, while data from the later time as the test set. Tab. 2 lists the detailed information of these datasets. Furthermore, we organize experts to label the test sets using our tool (See § 4.2), combining job scheduling lists and manual verification. The anomaly ratio is derived by dividing the number of labeled anomalous samples by the total samples in the test set. Notably, these performance anomalies are not necessarily failures but may indicate potential inefficiencies or transient issues.

- **D1:** We collect data from one array in the cluster, comprising 1,294 nodes and 13,379 assigned jobs over one week, sampled at 15-second intervals. As shown in Fig. 4, we conduct a real-world statistical experiment that approximately 94.9% of these job segments have a duration of less than one day. The data collection period of one week is deemed sufficient for model training and validation. Each compute node is monitored for 3,014 metrics, covering multiple dimensions of node performance. Detailed information is listed in Tab. 3. As described in § 3.2, we perform cleaning and reduction on the per-core metrics, ultimately ending up with 82 metrics in total.
- **D2:** We collect 30 nodes and 1,430 assigned jobs over 8 days sampled at an interval of 15 seconds. A total of 773 corresponding monitoring metrics are initially collected, and 116 metrics are finalized.

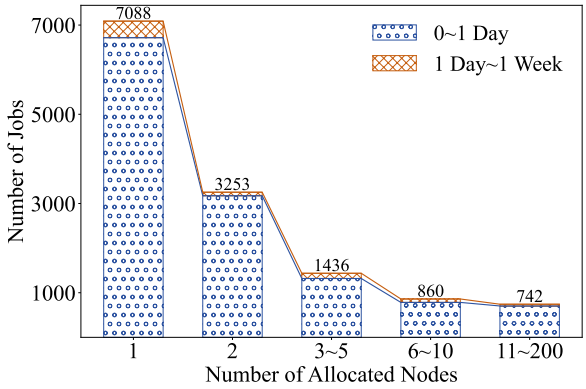

Figure 4: The distribution of jobs for nodes.

We chose not to use publicly available datasets because they lack the integration of job scheduling lists and monitoring metrics for nodes, which is essential for our objectives. The Antarex[2] dataset only provides system-level performance metrics. While the Prodigy[3] dataset covers nodes, it only provides intermediate files that have been processed by models, rather than raw data.

4.1.2 Baseline Methods. To evaluate the effectiveness of *NodeSentry*, we compare it with the four advanced baseline methods: Prodigy [4], RUAD [30], ExaMon [10], and ISC'20 [32]. We do not experiment with supervised learning methods. TPDS'18 [42] and ALBADross [3] employ machine learning classifiers, which introduce disparities in model complexity and supervision type. Importantly, DeepHYDRA [37], Proctor [5], and ExaMon [10] utilize semi-supervised methods. DeepHYDRA [37] and Proctor [5] heavily rely on their supervised component, whereas ExaMon [10] is derived from a comprehensive analysis that integrates both the supervised and unsupervised components. To ensure a fair comparison, we exclusively select the unsupervised methods employed in ExaMon [10] for evaluation. Furthermore, we do not select anomaly detection methods designed explicitly for non-HPC systems (*e.g.*, microservice systems or web systems), as instances in these systems have fixed tasks during runtime, and exhibit regular and periodic data patterns.

We configure the parameters of all these methods. Specifically, for parameter settings that are not specific to a particular dataset, we use the same configurations mentioned in the respective papers. For parameter settings that are dataset-specific, we adjust them according to the ranges provided in the respective papers or to our datasets.

[2] https://zenodo.org/records/2553224
[3] https://zenodo.org/records/8079388

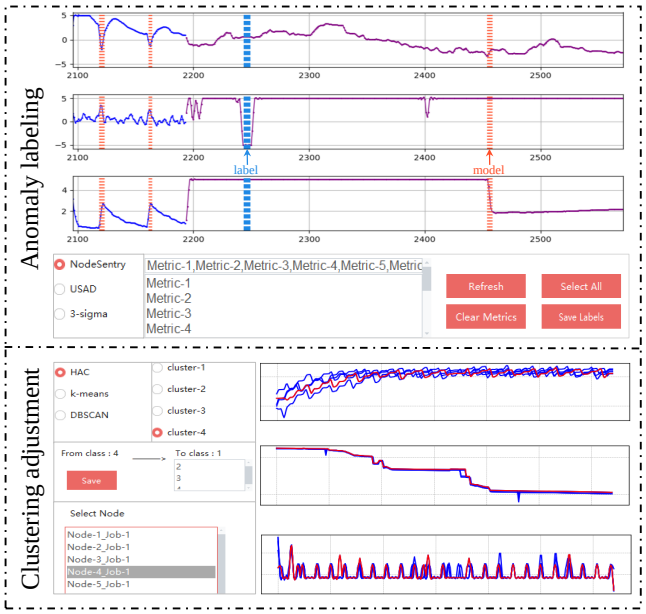

Figure 5: The interface of the labeling tool.

4.1.3 Implementation. All experiments are conducted on an offline validation platform with the following configuration: We implement *NodeSentry* and baseline methods using Python 3.8.10, PyTorch 2.0.1, and scikit-learn 1.1.1, running on a dedicated server equipped with 64 × Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, two NVIDIA(R) Tesla(R) V100S, and 187 GB RAM.

4.1.4 Evaluation Metrics. We use the standard point-wise anomaly detection metrics Precision, Recall, Area Under the Curve (AUC), and F1-score to evaluate the performance of all methods. Furthermore, in light of the multiple compute nodes present within our dataset, we average the Precision, Recall, and AUC across each node, with the F1-score being derived from the average Precision and Recall. Considering two practical considerations from operational experience, we employ an adjustment strategy widely utilized in previous studies [3–5, 10, 30, 32]: 1) Assuming the presence of actual continuous anomaly timestamps, we consider them as accurately detected if the method identifies any anomaly within those timestamps. 2) We deliberately exclude anomalies from the initial and terminal 1-minute intervals of each pattern transition, in which some metrics may significantly deviate from expected values.

4.2 A Labeling Tool for Experiment

Despite being a fully unsupervised anomaly detection method that operates without the need for labeling data, the verification of its accuracy still necessitates ground truth. Considering the high cost, difficulty, absence of distinct labeling standards, and challenges associated with handling high-dimensional metrics using existing labeling tools, we develop a graphical user interface-based tool allowing operators to adjust clusters and label anomalies in MTS. This tool is implemented in Python with Tkinter and Matplotlib, with the codebase spanning around 1,600 lines. Fig. 5 depicts the primary interface, encompassing the following functionalities:

1) It facilitates the selection of displaying or concealing specific metric dimensions, supports visualization, dragging, and horizontal/vertical zooming of MTS, and distinguishes diverse node states, thereby affording operators an enhanced perspective.
2) It enables operators to label or cancel anomalous intervals by specifying the start and end time intervals, saving them as anomalies. To alleviate the workload, we integrate multiple anomaly detection methods (*e.g.*, statistical methods and deep learning methods) to aid in labeling.
3) It incorporates built-in clustering methods and provides visualizations of the data distribution. Moreover, it facilitates dynamic adjustment of clusters and updates the centroids of each cluster.

We adopt a comprehensive approach based on job scheduling lists and manual verification by operators to avoid mislabeling and omissions. In particular, the eventual failure of a job cannot be solely attributed to performance anomalies in compute nodes [10]. It might be caused by code errors or insufficient storage resources. Additionally, performance anomalies in nodes may manifest before the job failure [13]. This principle also applies to the idle waiting state. Thus, real-time anomaly detection for compute nodes becomes imperative to ensure precise labeling. By detecting anomalies during job execution, we can proactively terminate the job to prevent any further abnormal propagation. When anomalies arise during idle waiting, operators can implement measures for isolation and mitigation.

4.3 Overall Performance (RQ1)

The performance of different methods is shown in Tab. 4. Compared to all baseline methods, *NodeSentry* achieves an impressive 0.876 and 0.891 F1-score, a relative improvement of 179.04% and 167.57% compared to the second place, respectively. The strategy of capturing and learning the pattern characteristics of nodes in different jobs through coarse-grained clustering and fine-grained model sharing has contributed to the success of *NodeSentry*. Other methods [4, 10, 30, 32] ignore the differences in patterns and sub-patterns. The specific reasons for each baseline's poor performance are discussed in detail in § 6. In contrast, *NodeSentry*'s tailored approach is sensitive to the nuances of node behavior and enhances anomaly detection performance by dynamically and adaptively selecting models in real time.

We analyze the complexity by comparing the time required for offline training with the average time to detect each node. *NodeSentry*'s anomaly detection efficiency is satisfactory, with the average time to detect whether each time point is anomalous not exceeding 2 milliseconds. This latency is well within the acceptable range for anomaly detection. Notably, ISC 20 [32] has the lowest training overhead, due to its use of Bayesian Gaussian Mixture Models (BGMM) clustering rather than deep learning models, resulting in its poorest performance. Compared with the other deep learning baseline methods, *NodeSentry* demonstrates the best training efficiency, with a relative improvement of 77.93% and 13.45% over the second-place method.

4.4 Ablation Study (RQ2)

To demonstrate the effectiveness of the key components in *NodeSentry* (*i.e.*, segment clustering and model sharing), we perform

Table 4: Effectiveness of anomaly detection on different methods.

Method	D1						D2					
	Precision	Recall	AUC	F1-score	Offline	Online	Precision	Recall	AUC	F1-score	Offline	Online
NodeSentry	**0.840**	**0.915**	**0.964**	**0.876**	1.06 day	2.47 s	**0.884**	**0.897**	**0.923**	**0.891**	27.21 min	2.31 s
Prodigy [4]	0.227	0.132	0.571	0.167	4.79 day	9.52 s	0.157	0.271	0.622	0.199	31.44 min	6.28 s
RUAD [30]	0.323	0.306	0.629	0.314	18.94 day	7.54 s	0.403	0.284	0.659	0.333	6.69 h	8.46 s
ExaMon [10]	0.203	0.217	0.586	0.210	7.95 day	**0.67 s**	0.407	0.216	0.612	0.282	3.35 h	**1.09 s**
ISC 20 [32]	0.026	0.154	0.557	0.045	**1.64 h**	7.35 s	0.006	0.103	0.50	0.012	**2.01 min**	8.81 s

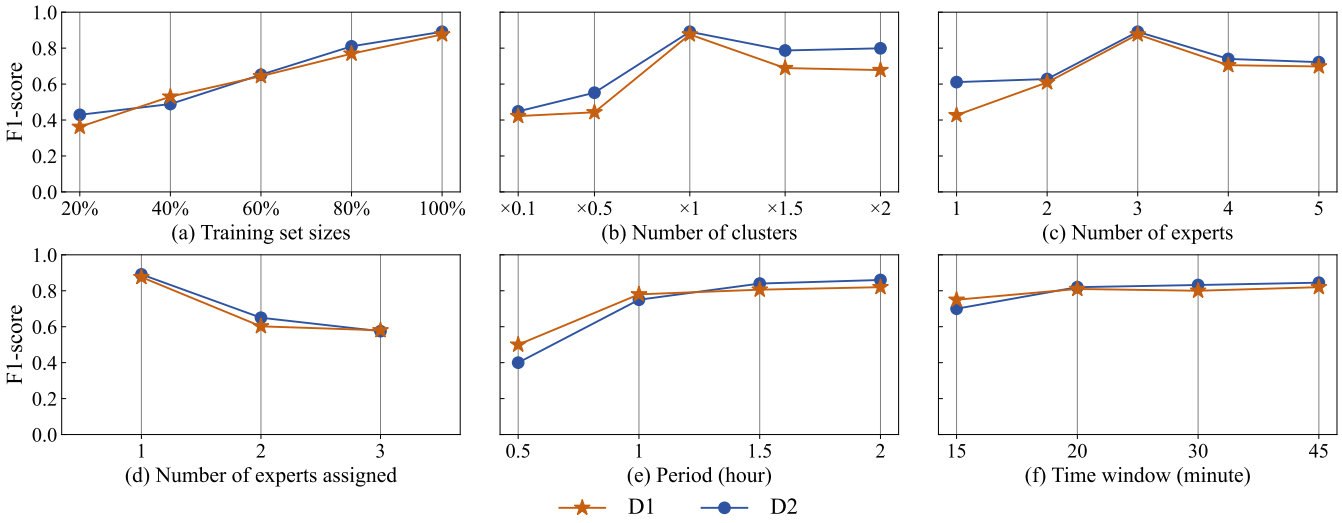

Figure 6: Effectiveness of *NodeSentry* under different hyperparameter settings.

Table 5: Performance comparison of different components.

Dataset	Method	Precision	Recall	AUC	F1-score
D1	*NodeSentry*	**0.840**	**0.915**	**0.964**	**0.876**
	C1	0.313	0.290	0.635	0.301
	C2	0.370	0.504	0.727	0.427
	C3	0.814	0.696	0.815	0.751
	C4	0.451	0.491	0.730	0.470
	C5	0.381	0.376	0.676	0.378
D2	*NodeSentry*	**0.884**	**0.897**	**0.923**	**0.891**
	C1	0.396	0.328	0.675	0.359
	C2	0.610	0.613	0.798	0.611
	C3	0.762	0.798	0.862	0.780
	C4	0.617	0.583	0.778	0.599
	C5	0.571	0.451	0.725	0.504

ablation experiments on two datasets, creating five variants **C1-C5**. 1) **C1** removes coarse-grained clustering and uses only a single model. 2) **C2** randomly selects segments to train the same number of models to replace the representative segments resulting from the clustering process. 3) **C3** cuts the segments of different lengths into equal lengths. 4) **C4** eliminates the practice of differentiating between segments within the positional encoding. 5) **C5** replaces the sparse MoE layer with the dense FFN layer.

Tab. 5 illustrates the enhanced performance of *NodeSentry* across various application scenarios, surpassing the performance of all previously mentioned alternatives. This highlights the crucial contribution of each component in attaining peak performance. Training a solitary model (**C1**) or employing an ensemble of models selected haphazardly (**C2**) forfeits the capacity to segment data into coherent subsets, thereby impeding the ability to hone in on the unique attributes of disparate data segments. The failure to leverage the structural information within the data precipitates a degradation in performance. Furthermore, both coarse-grained clustering and fine-grained model sharing constitute efficacious strategies for anomaly detection. The segments of different lengths contain disparate quantities of information, and treating these uniformly can precipitate an imbalance in the representation of job information (**C3**). Employing this variant can adversely affect both the effectiveness of anomaly detection and the operational efficiency of the model. With the model-sharing module, positional encoding enhances the ability to discern positional information across heterogeneous segments (**C4**). In contrast, the substitution of the sparse MoE layer with the dense FFN layer (**C5**) erodes the adaptability and versatility. This is attributable to the fact that the MoE layer is inherently more adept at managing the diversity and intricacy of data, a proficiency that the FFN layer is typically deficient in.

4.5 Incremental Training (RQ3)

To evaluate the influence of training set sizes and the effectiveness of incremental training, we discern a robust positive correlation between training set size and model performance. We conduct the

experiments starting from the minimal training set and incrementally increasing the size until reaching the full training set. Each size is derived via stratified random sampling from the full dataset, ensuring the randomness and impartiality essential for experimental validity. Model training adheres to uniform parameter settings to preclude confounding influences from extraneous variables.

As shown in Fig. 6 (a), the experimental findings reveal that the performance is significantly diminished with smaller training set sizes. This phenomenon may arise from the inability of a smaller training set to furnish sufficient data to cultivate a robust model, consequently resulting in *NodeSentry*'s subpar detection of new patterns. *NodeSentry*'s performance exhibits a marked enhancement with an escalation in training set size, a testament to its sophisticated structural design and judicious methodological choices. In practical scenarios where only limited data are available, the performance of anomaly detection can be optimized through an incremental training pipeline, as introduced in § 3.5.

4.6 Hyperparameters Sensitivity (RQ4)

We discuss the influence of six hyperparameters of *NodeSentry*. Details of the training set sizes can be found in § 4.5. Fig. 6 illustrates the impact of different hyperparameter settings on the F1-score.

1) The number of clusters in coarse-grained clustering. *NodeSentry*'s performance is significantly impaired when it falls below an optimal number, yet it stabilizes once this threshold is surpassed. As described in § 3.3, *NodeSentry* exhibits an innate capacity to autonomously identify the most productive cluster count, thereby maintaining peak performance without compromise.
2) The number of experts for MoE. An insufficient number of experts could lead to an incomplete representation of the data's salient features, while an excess might precipitate overfitting, thereby diminishing the generalizability. *NodeSentry* performs optimally when the number of experts is set to 3.
3) The number of experts assigned to each token. The specialization of each expert in identifying sub-pattern variations suggests that consolidating their outputs could lead to unwarranted complexity and potential inaccuracies. Consequently, *NodeSentry* performs best when each token is assigned to a single expert.
4) The period for pattern matching. A shorter period does not capture enough contextual information, thus reducing detection accuracy. Our experiments show that 1 hour is recommended for general use, with longer periods reserved for high-accuracy applications.
5) The time window for threshold selection. *NodeSentry* exhibits robustness in threshold selection, adapting well to varying window lengths. However, to ensure computational efficiency and model stability, shorter time windows (*e.g.*, 15 or 20 minutes) are recommended, as they reduce computational complexity without compromising performance.

5 Discussion

5.1 Deployment

NodeSentry was deployed on a dedicated node equipped with an 8-core, 64-threaded processor and 128 GB of RAM, designed to monitor and analyze a production environment cluster comparable in size and configuration to D2. During the deployment phase,

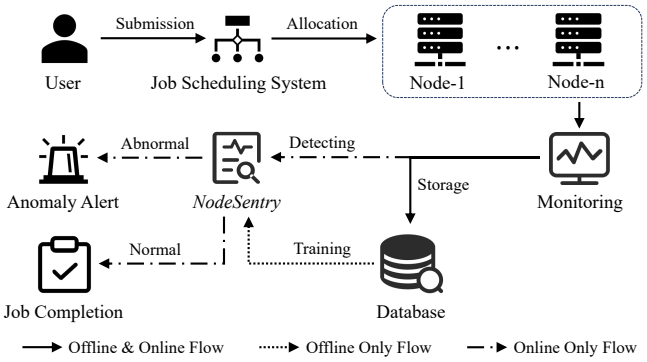

Figure 7: The workflow of *NodeSentry*.

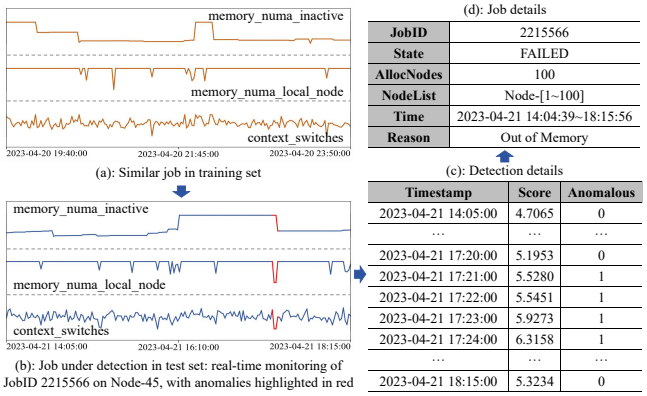

Figure 8: Case study of an out-of-memory case.

molecular dynamics simulations were executed using a Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [19] on this cluster. At the same time, failure injection scenarios (*e.g.*, disk full, memory exhaustion, and CPU overload) were systematically introduced via the ChaosBlade[4] toolkit to validate system robustness. Over a continuous one-month evaluation period, *NodeSentry* demonstrated efficient operational performance, completing pattern matching for each hourly monitoring cycle in an average of 5.11 seconds and achieving real-time detection latency of 36 milliseconds per sampling point. *NodeSentry* exhibited strong anomaly detection capabilities, with precision and recall rates of 0.857 and 0.923, respectively, in identifying performance anomalies and injected failures.

As shown in Fig. 7, the deployment workflow integrates both offline and online operational modes. When a user submits a computational job, the Slurm job scheduler dynamically allocates resources across the HPC cluster based on user-defined parameters such as node count and runtime requirements. Concurrently, Prometheus[5] collects granular performance metrics from all nodes, storing this telemetry data in a time-series database to support offline model training. During the online phase, the collected data is passed to *NodeSentry* in real time for anomaly detection. Upon detecting anomalies, *NodeSentry* triggers prioritized alerts to operators.

5.2 Case Study

In this section, we conduct a case study on D1 to further demonstrate the working process of *NodeSentry*. As shown in Fig. 8, the memory-level failure in Node-45 led to anomalies in the metrics. *NodeSentry* matches the job pattern under detection with the most similar historical patterns and applies the learned model to detect anomalies. The difference between the raw and reconstructed MTS is calculated at each time point. *NodeSentry* detected the node anomaly 54 minutes before the job failure, allowing the operators to intervene early to prevent the job failure. Because memory-related metrics showed significant declines, insufficient memory was identified as the cause of the job failure. Compared with traditional methods, *NodeSentry* performs exceptionally well by effectively handling complex job patterns and sub-patterns, making it more suitable for the dynamic and large-scale nature of HPC systems.

5.3 Limitations and Threats

During online detection, the limitations of *NodeSentry* mainly concern the frequent job transitions of nodes. Timely and precise pattern matching is necessary to uphold model accuracy, while a certain duration of time is utilized. Nonetheless, experiments indicate that this temporal investment is considered acceptable.

Regarding the internal validity threats, we conduct rigorous repeated experimental tests and iteratively optimize the configuration of each module. The experimental results represent the average of multiple trials. *NodeSentry* also faces external validity threats. This field lacks publicly available and fair datasets. We validate *NodeSentry* using a large-scale supercomputer system. However, this may not fully represent all HPC systems. Our validation centered on CPU, but GPU compute units demonstrate comparable data characteristics and are equally subject to frequent task transitions. Nonetheless, we have confidence in the generality of *NodeSentry*.

6 Related Work

Various anomaly detection methods for nodes in HPC systems can be roughly divided into three groups: supervised methods, semi-supervised methods, and unsupervised methods.

Supervised methods. TPDS 18 [42] and ALBADross [3] utilize feature extraction and feature selection techniques, respectively. They then train a machine learning classifier to detect different performance anomalies. These methods require the use of labeled data for training that contains both normal and abnormal samples. However, they demand extensive domain knowledge and labeling efforts from experts, which is often labor-intensive. Therefore, these methods are almost impractical for large-scale HPC systems.

Semi-supervised methods. DeepHYDRA [37] combines Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and learning-based anomaly detection, introducing a supervised MSE-based loss function for semi-supervised training. Proctor [5] and ExaMon [10] employ Auto-Encoder (AE) for anomaly detection. Proctor [5] learns hidden layer features to detect anomalies through a supervised classifier. On the other hand, ExaMon [10] combines the results of reconstruction error and classification probability. Compared to supervised methods, they only require a relatively small amount of labeled data and a large amount of unlabeled data for training. However, this limitation restricts the modeling capability for abnormal samples.

Unsupervised methods. RUAD [30] utilizes Long Short-Term Memory (LSTM) cells, explicitly capturing temporal dependencies. RUAD [30] requires training specific deep models for each node, resulting in additional storage and scheduling requirements. ISC 20 [32], which employs BGMM and Mahalanobis distance to fit Gaussian distributions, is limited in its capability to effectively model the complex dynamics of MTS data using machine learning alone. Prodigy [4] is an anomaly detection framework based on Variational AE (VAE). ISC 20 [32] focuses on clustering the entire segments, while Prodigy [4] extracts features for subsequent detection processes. Most importantly, neither of them takes into consideration the intricate sub-patterns that can vary significantly across the same segment.

7 Conclusion

Given the inherent complexity of maintaining operational stability in HPC systems, we propose *NodeSentry*, an unsupervised MTS anomaly detection framework specifically designed for nodes in HPC systems. *NodeSentry* integrates coarse-grained segment clustering with fine-grained model sharing, enhancing its scalability and efficiency while significantly improving anomaly detection accuracy and generalization capabilities. Through extensive evaluation using data collected from production HPC systems, we demonstrate *NodeSentry*'s effectiveness in achieving high precision and recall for anomaly detection. Furthermore, to promote reproducibility, we have open-sourced *NodeSentry*'s codebase and introduced a novel clustering adjustment and anomaly labeling tool specifically designed for HPC systems.

Acknowledgments

This work is supported by the Advanced Research Project of China (31511010501), the National Natural Science Foundation of China (62272249, 62302244), and the Fundamental Research Funds for the Central Universities (XXX-63253249).

References

[1] 2025. sacct. https://slurm.schedmd.com/sacct.html. [Online].
[2] Mahdi Abavisani, Alireza Naghizadeh, Dimitris Metaxas, and Vishal Patel. 2020. Deep subspace clustering with data augmentation. *Advances in Neural Information Processing Systems* 33 (2020), 10360–10370.
[3] Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse Coskun. 2024. Runtime Performance Anomaly Diagnosis in Production HPC Systems Using Active Learning. *IEEE Transactions on Parallel and Distributed Systems* 35 (2024), 693–706.
[4] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[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] Marília Barandas, Duarte Folgado, Letícia Fernandes, Sara Santos, Mariana Abreu, Patrícia Bota, Hui Liu, Tanja Schultz, and Hugo Gamboa. 2020. TSFEL: Time series feature extraction library. *SoftwareX* 11 (2020), 100456.
[7] Paul Bergmann, Kilian Batzner, Michael Fauser, David Sattlegger, and Carsten Steger. 2021. The MVTec anomaly detection dataset: a comprehensive real-world

[4] https://chaosblade.io/
[5] https://prometheus.io/

dataset for unsupervised anomaly detection. *International Journal of Computer Vision* 129, 4 (2021), 1038–1059.

[8] Maciej Besta, Marcel Schneider, Marek Konieczny, Karolina Cynk, Erik Henriksson, Salvatore Di Girolamo, Ankit Singla, and Torsten Hoefler. 2020. FatPaths: Routing in supercomputers and data centers when shortest paths fall short. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–18.

[9] Angela Bonifati, Francesco Del Buono, Francesco Guerra, and Donato Tiano. 2022. Time2Feat: learning interpretable representations for multivariate time series clustering. *Proceedings of the VLDB Endowment (PVLDB)* 16, 2 (2022), 193–201.

[10] Andrea Borghesi, Martin Molan, Michela Milano, and Andrea Bartolini. 2021. Anomaly detection and anticipation in high performance computing systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 739–750.

[11] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 536–547.

[12] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. 2022. Towards understanding the mixture-of-experts layer in deep learning. *Advances in neural information processing systems* 35 (2022), 23049–23062.

[13] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, Yongqiang Yang, and Michael R Lyu. 2022. Adaptive performance anomaly detection for online service systems via pattern sketching. In *Proceedings of the 44th international conference on software engineering*. 61–72.

[14] Yiqin Dai, Yong Dong, Kai Lu, Ruibo Wang, Wei Zhang, Juan Chen, Mingtian Shao, and Zheng Wang. 2022. Towards scalable resource management for supercomputers. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[15] Wangxiang Ding, Wenzhong Li, Zhijie Zhang, Chen Wan, Jianhui Duan, and Sanglu Lu. 2022. Time-varying Gaussian Markov random fields learning for multivariate time series clustering. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2022), 11950–11966.

[16] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[17] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 120–134.

[18] Zilong He, Pengfei Chen, and Tao Huang. 2022. Share or not share? towards the practicability of deep models for unsupervised anomaly detection in modern online systems. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 25–35.

[19] Dan Huang, Zhenlu Qin, Qing Liu, Norbert Podhorszki, and Scott Klasky. 2022. Identifying challenges and opportunities of in-memory computing on large hpc systems. *J. Parallel and Distrib. Comput.* 164 (2022), 106–122.

[20] Kevin Huck and Allen Malony. 2023. ZeroSum: User Space Monitoring of Resource Utilization and Contention on Heterogeneous HPC Systems. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 685–695.

[21] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1018–1031.

[22] Minqi Jiang, Songqiao Han, and Hailiang Huang. 2023. Anomaly detection with score distribution discrimination. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 984–996.

[23] Wenzhao Jiang, Jindong Han, Hao Liu, Tao Tao, Naiqiang Tan, and Hui Xiong. 2024. Interpretable cascading mixture-of-experts for urban traffic congestion prediction. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5206–5217.

[24] Boris Kozinsky, Albert Musaelian, Anders Johansson, and Simon Batzner. 2023. Scaling the leading accuracy of deep equivariant models to biomolecular simulations of realistic size. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[25] Xuan-May Le, Ling Luo, Uwe Aickelin, and Minh-Tuan Tran. 2024. Shapeformer: Shapelet transformer for multivariate time series classification. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1484–1494.

[26] Dongha Lee, Seonghyeon Lee, and Hwanjo Yu. 2021. Learnable dynamic temporal pooling for time series classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8288–8296.

[27] Dongwen Li, Shenglin Zhang, Yongqian Sun, Yang Guo, Zeyu Che, Shiqi Chen, Zhenyu Zhong, Minghan Liang, Minyi Shao, Mingjie Li, Shuyang Liu, Yuzhi Zhang, and Dan Pei. 2023. An Empirical Analysis of Anomaly Detection Methods for Multivariate Time Series. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 57–68.

[28] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices* 53, 2 (2018), 419–431.

[29] Ruiying Lu, YuJie Wu, Long Tian, Dongsheng Wang, Bo Chen, Xiyang Liu, and Ruimin Hu. 2023. Hierarchical vector quantized transformer for multi-class unsupervised anomaly detection. *Advances in Neural Information Processing Systems* (2023), 8487–8500.

[30] 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.

[31] Nicholas Monath, Kumar Avinava Dubey, Guru Guruganesh, Manzil Zaheer, Amr Ahmed, Andrew McCallum, Gokhan Mergen, Marc Najork, Mert Terzihan, Bryon Tjanaka, et al. 2021. Scalable hierarchical agglomerative clustering. In *Proceedings of the 27th ACM SIGKDD Conference on knowledge discovery & data mining*. 1245–1255.

[32] 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.

[33] George Papadimitriou and Dimitris Gizopoulos. 2023. Silent data corruptions: Microarchitectural perspectives. *IEEE Trans. Comput.* 72, 11 (2023), 3072–3085.

[34] Armin Danesh Pazho, Ghazal Alinezhad Noghre, Arnab A Purkayastha, Jagannadh Vempati, Otto Martin, and Hamed Tabkhi. 2023. A survey of graph-based deep learning for anomaly detection in distributed systems. *IEEE Transactions on Knowledge and Data Engineering* 36, 1 (2023), 1–20.

[35] Efe Sencan, Yin-Ching Lee, Connor Casey, Benjamin Schwaller, Vitus J Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K Coskun. 2025. Refine: Robust Unsupervised Anomaly Detection for Production HPC Systems. (2025).

[36] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. 2024. ScheMoE: An Extensible Mixture-of-Experts Distributed Training System with Tasks Scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 236–249.

[37] Franz Kevin Stehle, Wainer Vandelli, Felix Zahn, Giuseppe Avolio, and Holger Fröning. 2024. DeepHYDRA: A Hybrid Deep Learning and DBSCAN-Based Approach to Time-Series Anomaly Detection in Dynamically-Configured Systems. In *Proceedings of the 38th ACM International Conference on Supercomputing*. Springer, 272–285.

[38] Ming Sun, Ya Su, Shenglin Zhang, Yuanpu Cao, Yuqing Liu, Dan Pei, Wenfei Wu, Yongsu Zhang, Xiaozhou Liu, and Junliang Tang. 2021. CTF: Anomaly detection in high-dimensional time series with coarse-to-fine model transfer. In *IEEE INFOCOM 2021-IEEE conference on computer communications*. IEEE, 1–10.

[39] Yongqian Sun, Minghan Liang, Zeyu Che, Dongwen Li, Tinghua Zheng, Shenglin Zhang, Pengtian Zhu, Yuzhi Zhang, and Dan Pei. 2023. Efficient Multivariate Time Series Anomaly Detection Through Transfer Learning for Large-Scale Web Services. In *2023 IEEE International Conference on Web Services (ICWS)*. IEEE, 145–151.

[40] Yi Tay, Dara Bahri, Donald Metzler, Da-Cheng Juan, Zhe Zhao, and Che Zheng. 2021. Synthesizer: Rethinking self-attention for transformer models. In *International conference on machine learning*. PMLR, 10183–10192.

[41] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. 2022. Tranad: Deep transformer networks for anomaly detection in multivariate time series data. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1201–1214.

[42] 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. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2018), 883–896.

[43] Steven Euijong Whang, Yuji Roh, Hwanjun Song, and Jae-Gil Lee. 2023. Data collection and quality challenges in deep learning: A data-centric ai perspective. *The VLDB Journal* 32, 4 (2023), 791–813.

[44] Justin Whitt. 2022. Frontier Testing and Tuning Problems Downplayed by Oak Ridge. *InsideHPC* (2022). https://insidehpc.com/2022/10/frontier-testing-and-tuning-problems-downplayed-by-oak-ridge/ Accessed: 2024-04-21.

[45] Yuan Xie, Shaohan Huang, Tianyu Chen, and Furu Wei. 2023. Moec: Mixture of expert clusters. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 13807–13815.

[46] Bo Yang, Xiao Fu, Nicholas D Sidiropoulos, and Mingyi Hong. 2017. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In *international conference on machine learning*. PMLR, 3861–3870.

[47] Yiyuan Yang, Chaoli Zhang, Tian Zhou, Qingsong Wen, and Liang Sun. 2023. Dcdetector: Dual attention contrastive representation learning for time series anomaly detection. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3033–3045.

[48] Nan Zhang and Shiliang Sun. 2022. Multiview unsupervised shapelet learning for multivariate time series clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 4 (2022), 4981–4996.

[49] Shenglin Zhang, Dongwen Li, Zhenyu Zhong, Jun Zhu, Minghan Liang, Jiexi Luo, Yongqian Sun, Ya Su, Sibo Xia, Zhongyou Hu, et al. 2022. Robust system instance clustering for large-scale web services. In *Proceedings of the ACM Web Conference 2022*. ACM, 1785–1796.

[50] 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 *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 162–168.

[51] Zijian Zhang, Shuchang Liu, Jiaao Yu, Qingpeng Cai, Xiangyu Zhao, Chunxu Zhang, Ziru Liu, Qidong Liu, Hongwei Zhao, Lantao Hu, et al. 2024. M3oE: Multi-Domain Multi-Task Mixture-of Experts Recommendation Framework. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 893–902.

[52] Yue Zhao, Guoqing Zheng, Subhabrata Mukherjee, Robert McCann, and Ahmed Awadallah. 2023. Admoe: Anomaly detection with mixture-of-experts from noisy labels. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 4937–4945.

[53] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*. 11106–11115.

[54] Kaiyang Zhou, Yongxin Yang, Yu Qiao, and Tao Xiang. 2021. Domain adaptive ensemble learning. *IEEE Transactions on Image Processing* 30 (2021), 8008–8018.

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

This paper aims to address unsupervised anomaly detection for compute nodes in high-performance computing (HPC) systems by proposing a novel framework that combines coarse-grained pattern clustering with fine-grained model optimization, while providing tools to improve interpretability and reproducibility.

C_1 A hybrid anomaly detection framework integrating Hierarchical Agglomerative Clustering (HAC) for coarse-grained pattern grouping, followed by a Mixture of Expert (MoE) architecture for fine-grained model sharing and sub-pattern adaptation. This design reduces training overhead while achieving state-of-the-art F1-scores.

C_2 An open-source toolkit containing 1) a clustering adjustment interface for validating HAC results, and 2) an interactive anomaly labeling tool for multivariate time series in HPC contexts. This toolkit enables transparent pattern analysis and ground-truth refinement.

A.2 Computational Artifacts

To support the reproducibility, we provide two computational artifacts that encapsulate the core implementation and analytical tools. These artifacts enable validation of the proposed method and facilitate further exploration of HPC anomaly detection scenarios.

A_1 https://zenodo.org/records/16737328
A_2 https://zenodo.org/records/16737389

The following table explicitly maps the paper's artifacts to their corresponding contributions and reproducible elements. Each artifact directly generates or validates specific experimental results and visualizations.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Tables 4-5 Figures 6
A_2	C_2	Figure 5

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact provides the full implementation of the hybrid framework described in C_1, which unifies coarse-grained clustering and fine-grained model adaptation.

Expected Results

The experimental results conclusively demonstrate the effectiveness of our proposed framework in addressing the core challenges of HPC anomaly detection. The superior performance validates Contribution C_1, showing that our hierarchical approach combining coarse clustering and fine-grained adaptation successfully captures macro-level patterns and micro-level variations in node behaviors. Ablation studies reveal critical dependencies, proving the necessity of each component.

Expected Reproduction Time (in Minutes)

The complete experimental workflow can be reproduced within approximately 24 minutes using open-source data and standard computing resources.

Artifact Setup (incl. Inputs)

Hardware. The experimental environment is deployed on servers running Linux kernel version 5.4.0 and Ubuntu 18.04 operating system. Detailed hardware specifications, including computational resources and network configurations, are comprehensively documented in Section 4.1. The system architecture supports distributed monitoring across seven physical nodes, with metric collection mechanisms optimized for this server configuration.

Software. The implementation leverages Python 3.8 as the primary programming language, with critical dependencies including ChaosBlade for fault injection and Prometheus for metric collection. LAMMPS serves as the benchmark workload for system validation. All required software packages and version-specific libraries are enumerated in the requirements.txt file within the project repository. The software stack has been verified for compatibility with the specified kernel version and Ubuntu distribution.

Datasets / Inputs. A representative subset of experimental data from *NG-Tianhe*, comprising 138-dimensional monitoring metrics collected from 17 distinct benchmark jobs across seven compute nodes, has been made publicly accessible through the repository. The dataset captures system behaviors under both normal operation and 21 artificially induced fault scenarios, generated through controlled ChaosBlade injections during LAMMPS executions.

Installation and Deployment. For environment reproducibility, users should first establish a Python 3.8 virtual environment on a Linux 5.4.0/Ubuntu 18.04 system. Execute sequential commands after environment activation:

T1: *pip install –upgrade pip*
T2: *pip install -r requirements.txt*

The repository provides architecture-specific compilation guidelines and dependency resolution details in its *README*. To ensure metric collection integrity, special attention is required when configuring Prometheus exporters and ChaosBlade controllers.

Artifact Execution

The artifact's execution pipeline comprises three sequential phases: coarse-grained clustering, fine-grained shared model training, and anomaly detection. During the initial clustering phase, feature vectors extracted from input data undergo unsupervised grouping, producing cluster centroids and metric weight distributions for each identified class. Subsequent model training employs cluster-specific configurations, where shared architectural components are optimized through distributed parameter updates while preserving

cluster-wise specialization. The detection phase computes anomaly scores at each temporal sampling point through multivariate pattern analysis across all data replicas.

Experimental configurations utilize a dataset spanning 7 compute nodes executing 17 distinct workloads, with each observation capturing 138-dimensional vectors. Implementation specifications include batch processing of 50 samples per iteration, input windowing of 20 temporal steps, and model optimization across 30 training epochs using a learning rate of 0.00015. The architecture integrates a 3-layer Transformer encoder with 3 parallel attention heads, and a MoE layer that selectively combines outputs from 3 domain-specific experts through top-1 gating activation.

Artifact Analysis (incl. Outputs)

The execution pipeline's output structure is governed by parameters defined in the *config.yml*. During coarse-grained clustering operations, extracted feature vectors are stored in the *normalized*, while resultant cluster assignments persist in *cluster*. Cluster centroids and associated metric weight distributions for each category are systematically archived within date-stamped subdirectories of the *center_feature_date*, following automatic per-class directory generation. In the model specialization phase, class-specific trained models are serialized to the *model_dir* repository, with corresponding anomaly detection outputs (containing temporal anomaly scores and diagnostic metadata) being written to the *result_dir* destination.

B.2 Computational Artifact A_2

Relation To Contributions

This artifact provides the functional implementation of the open-source toolkit described in C_2, directly realizing its dual objectives of cluster validation and interactive labeling.

Expected Results

The toolkit demonstrates how its interactive capabilities translate theoretical contributions into operational improvements. The visualization interface enables efficient pattern discovery by allowing operators to explore multivariate time series characteristics intuitively. At the same time, the semi-automated labeling system combines algorithmic preprocessing with human expertise to generate reliable annotations. Most importantly, the dynamic clustering adjustment feature creates a feedback loop between unsupervised detection and expert knowledge, allowing continuous refinement of anomaly labels and model performance. These outcomes collectively validate the tool's role in making unsupervised methods practical for production environments by bridging the gap between automated detection and human oversight.

Expected Reproduction Time (in Minutes)

The complete experimental reproduction requires 25-30 minutes under baseline benchmark measurements, partitioned as follows: Dependency resolution through automated package management consumes approximately 10 minutes (manual CUDA toolkit configuration and PyTorch compilation may require additional 5-15 minutes based on hardware specifics); Execution phases typically complete within 10-15 minutes, with anomaly detection and label generation for 20-node clusters (10 metrics × 500 temporal samples) requiring approximately 8 minutes, while clustering operations generally conclude within 3 minutes. Actual duration scales proportionally with input dimensionality and selected algorithmic implementations.

Artifact Setup (incl. Inputs)

Hardware. The tool is compatible with standard Windows/macOS workstations. While CPU-only execution is supported, GPU acceleration significantly reduces runtime for large datasets.

Software. The implementation uses Python 3.8+ with dependencies. Key libraries are pinned in *requirements.txt*.

Datasets / Inputs. The public repository includes synthetic time-series data in *node_data/* mimicking HPC node behaviors. Each CSV file represents a node's metrics over 500 timestamps, formatted as *timestamp, metric1, ..., metric10*. Real-world users should replace these with their data, ensuring column consistency across files. Configuration files (*metric_used.txt*, *time_scope.txt*) define target metrics (e.g., CPU usage, memory pressure) and time windows, enabling flexible adaptation to diverse monitoring frameworks.

Installation and Deployment. The process comprises four key steps:

T1: *git clone* A_2

T2: Create a Python 3.8+ virtual environment

T3: *pip install -r requirements.txt*

T4: Place custom node CSV files in *node_data/* and update *metric_used.txt* and *time_scope.txt*.

Full deployment guidelines are documented in the *README*.

Artifact Execution

The analytical tool is initialized through execution of *python main.py* on Windows or macOS. The graphical interface supports workflow orchestration via three core modules:

T1: Algorithm selection from the *reference_cluster*, with hyperparameter tuning through interactive sliders.

T2: Model specification including statistical criterion or deep neural architectures via the *reference_models*.

T3: Plot manipulation through mouse wheel zoom controls, click-and-drag panning operations, and anomaly annotation through rectangular region selection via primary mouse button activation. Cluster membership adjustments following manual node reassignment are persistently archived in designated configuration files to facilitate iterative analytical workflows.

Artifact Analysis (incl. Outputs)

Clustering results are stored in two files: *config_files/cluster_result.txt* (raw algorithmic outputs) and *cluster_adjust.txt* (user-modified groupings). Anomaly labels are saved as per-node CSV files in the *labels/* directory, while *annotation_history.txt* caches global annotation metadata.

For visualization, the *PlotCanvas2* tool enables metric-specific time-series plotting with anomaly highlights, and *MultiCanvas* supports side-by-side cluster comparisons. Users can validate results by cross-referencing raw data, cluster assignments, and annotated anomalies, ensuring transparency and reproducibility.