Configurable Containerized Testbed for Dataset Building for Anomaly Detection in Data Processing Infrastructure using Machine Learning

Minh Hieu Nguyen, Dmitry Ilin

Abstract—**The article presents a fully container-based experimental testbed for producing high-resolution monitoring data with controlled anomalies, allowing for the methodical assessment of machine learning techniques for infrastructure-level anomaly detection. The platform creates synchronized application, database, and system-level metrics at 1 Hz across 18 workload configurations by integrating a FastAPI-PostgreSQL service, workload generation using Locust, and telemetry collection via Prometheus and cAdvisor. Deterministic fault-triggering mechanisms inject two realistic anomaly types, CPU saturation and database latency, online during workload execution. A comprehensive dataset is constructed and statistically analyzed, revealing distinct resource utilization patterns associated with each anomaly. Eight machine learning models are evaluated, including four supervised classifiers (logistic regression, random forest, XGBoost, and lightGBM) and four unsupervised anomaly detection methods (isolation forest, one-class SVM, local outlier factor, and PCA-based reconstruction error). The supervised models show superior detection performance, but the unsupervised approaches are less sensitive to even the smallest degradation of resources. The proposed infrastructure and dataset provide a reproducible basis for investigating performance anomalies in microservice contexts and enable future research on automated monitoring and anomaly detection.**

Keywords—**Containerized systems; performance anomaly detection; monitoring dataset; machine learning; workload-driven fault injection.**

I. Introduction

Modern cloud and data-processing infrastructures continuously generate multivariate monitoring data to track system health and detect failures. Anomaly detection in such environments remains challenging due to complex interactions among services, workloads, and shared resources. Observability signals are usually high-dimensional, and anomalies can show up as spikes, slow drifts, or coordinated changes across several metrics. Prior surveys show that detection performance varies significantly across datasets, influenced by sampling resolution, preprocessing choices, and anomaly definitions [1-3]. These results emphasize that rather than depending exclusively on simplified public standards, anomaly-detection techniques should be evaluated in actual operating situations [1,3].

Obtaining high-quality labeled monitoring data from production systems is difficult because real anomalies are rare, labels are often incomplete or reconstructed after incidents, and privacy constraints limit data sharing [1,4]. To overcome data scarcity, many studies rely on synthetic time series labeled with heuristic rules or datasets from other domains, including industrial control systems or satellite telemetry [2,5]. However, these datasets fail to adequately capture the coupled behavior and failure dynamics of multi-tier microservice infrastructures. Poorly designed synthetic data may further oversimplify real failure modes, leading to misleading evaluations [6].

The necessity to evaluate anomaly-detection techniques throughout the entire monitoring pipeline, from metric collection and labeling to model evaluation, is emphasized by recent AIOps research [1,7]. Nevertheless, most public benchmarks lack realistic workload variation, complete monitoring stacks, or system-level fault injection. Because of this, researching how resource problems like CPU contention or database saturation spread across metrics and microservice components is still challenging [2,7].

A useful method for examining system behavior in controlled yet realistic settings is to use experimental testbeds. Deterministic perturbations can safely simulate actual operational failures, as shown by fault injection and chaos engineering [8-11]. There are still few publicly accessible platforms that concentrate exclusively on infrastructure-level monitoring measures, such as CPU utilization, memory consumption, request throughput, and database activity, despite the fact that testbeds have been proposed in a variety of situations [8,9,12]. Furthermore, the validity of benchmarking results may be jeopardized by abnormal patterns that do not coincide with actual failure mechanisms [12,13].

To address these gaps, this study introduces a reproducible, container-based experimental testbed for generating monitoring datasets under controlled workloads and system-level anomalies. The testbed emulates a lightweight microservice architecture comprising an HTTP API, a PostgreSQL database, a load generator, and a monitoring stack. Predefined configurations vary in

Manuscript received February 26, 2026.

Nguyen Minh Hieu, MIREA – Russian Technological University, Moscow, Russia (e-mail: nguen.m2@edu.mirea.ru).

Dmitry Ilin, MIREA – Russian Technological University, Moscow, Russia (e-mail: i@dmitryilin.com).

workload intensity and request profiles, while CPU saturation and induced database slowdowns inject anomalies at runtime. A labeled multivariate time series with genuine inter-metric interactions is the resultant dataset.

Using this dataset, four supervised and four unsupervised anomaly-detection models are systematically evaluated [14]. Standard classification metrics are used to evaluate performance, and SHAP-based feature analysis is used to determine important signs linked to database and CPU anomalies. The results provide a transparent and reproducible baseline for assessing anomaly-detection methods in microservice environments. This work makes three contributions:

- Creation of a replicable, open testbed that can produce infrastructure monitoring data under regulated load profiles and inject realistic real-time anomalies.
- Release of carefully selected training and test datasets, including full metric descriptions, anomaly labels, and scripts for data collection and workload orchestration.
- Systematic evaluation of supervised and unsupervised anomaly-detection techniques, along with interpretable analyses of feature importance using SHAP values.

This is how the rest of the paper is structured. Section II explains system architecture and the data generation pipeline. Section III presents the statistical analysis and important empirical findings. Section IV presents the machine learning approach and the experimental findings. Section V concludes with further research.

II. System Architecture and Data Generation Pipeline

A. Architecture Overview

As shown in Fig. 1, a containerized architecture controlled by Docker Engine and Docker Compose was used to construct the experimental platform. The API container, Locust container, Collector container, Prometheus/cAdvisor monitoring containers, and an additional Orchestrator container that handles controlled anomaly injection and a persistent dataset storage module constitute the system's five primary functional containers. This modular structure enables repeatable load testing, fine-grained monitoring, real-time data aggregation, and detailed anomaly characterization of resource consumption.

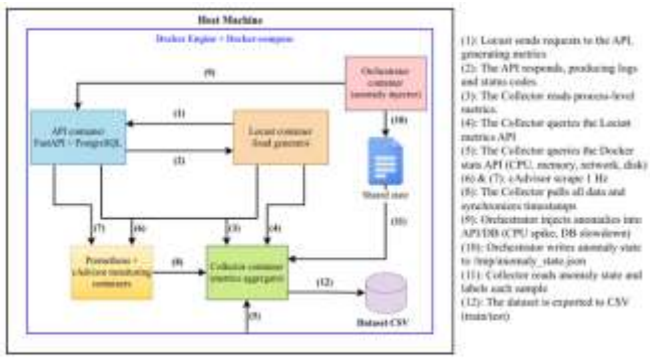
Host Machine
Docker Engine + Docker compose
Orchestrator container (anomaly injector)
API container FastAPI + PostgreSQL
Locust container (load generator)
Shared state
Prometheus + cAdvisor monitoring containers
Collector container (metrics aggregator)
Dataset CSV
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
(1): Locust sends requests to the API, generating metrics
(2): The API responds, producing logs and status codes.
(3): The Collector reads process-level metrics.
(4): The Collector queries the Locust metrics API
(5): The Collector queries the Docker stats API (CPU, memory, network, disk)
(6) & (7): cAdvisor scrape 1 Hz
(8): The Collector pulls all data and synchronizes timestamps
(9): Orchestrator injects anomalies into API/DB (CPU spike, DB slowdown)
(10): Orchestrator writes anomaly state to /tmp/anomaly_state.json
(11): Collector reads anomaly state and labels each sample
(12): The dataset is exported to CSV (train/test)

Fig. 1. Proposed containerized API-database monitoring architecture

The API container operates a FastAPI-based REST service with a PostgreSQL backend. It manages all incoming traffic from the load generator and exposes CRUD and file operation APIs. The API processes Locust's HTTP requests (1) and generates execution traces, which include latency measurements, status codes, and I/O activities (2). This container models the application tier commonly observed in microservice deployments [15,16]. The Locust container creates workload traffic by simulating concurrent clients running predefined workload profiles and continuously sends API requests (1) and records response statistics such as throughput, latency, and error rates. Locust's HTTP monitoring interface makes these performance metrics available, allowing the collector to periodically retrieve them (4). For assessing system behavior under dynamic workloads, the load generator thus offers adjustable stress conditions [17, 18]. The central aggregation module is a special collector container. It gathers heterogeneous signals from multiple monitoring sources: process-level API/database logs (3), Locust performance statistics (4), and infrastructure-level resource metrics retrieved from Docker's monitoring API (5). For training and testing, the collector exports the processed data to CSV files after synchronizing all streams at a one-second precision and combining the signals into a single time-series record (12). This integration ensures that application-level and system-level observations remain temporally aligned [19]. The Prometheus and cAdvisor monitoring containers give system telemetry. While Prometheus scrapes these metrics at a 1 Hz frequency, cAdvisor makes available container resource statistics such as CPU use, memory footprint, network throughput, and disk operations (6-7). This monitoring layer provides a fine-grained view of container performance attributes and records real-time variations in system resource utilization [20, 21]. The collector retrieves these measurements through Prometheus' API (8), enabling real-time synchronization with other performance indicators. Furthermore, an orchestrator container is responsible for introducing temporally scheduled controlled anomalies into the system. The orchestrator triggers fault events such as API-level CPU spikes or database slowdowns (9), writes anomalies into a shared state file (10), and exposes this information for downstream processing. To ensure accurate ground-truth annotation for machine learning studies, the collector reads the anomalous state (11) and labels each synchronized sample appropriately. Each container may operate independently thanks to the modular architecture, which also keeps data capture across the workload, application, and system levels synced. Orchestrator-driven anomalies, including controlled database slowdowns and CPU spikes in the API tier, add reproducible fault patterns that resemble production issues, increasing the dataset's realism through shared state. Consequently, the platform enables studies on anomaly detection in realistic failure situations as well as normal operation modeling. Keep in mind that the architecture diagram's numbering does not reflect the sequence of execution; rather, it merely shows the data flow.

B. Data Collection Pipeline

The dataset was produced using 18 workload profiles, consisting of 15 training profiles (P1-P15) and 3 testing profiles (T1-T3), as indexed by the profile_id field in the final dataset. Each profile defines a specific combination of user concurrency levels and API operation mixtures, which together determine the stress imposed on the containerized API-database system. All profiles were executed using the Locust load generator, which controls both the number of virtual users and the probabilistic distribution of CRUD operations during each run.

The two main dimensions that define the workload profiles are CRUD intensity, which controls the complexity of API interactions, and the number of concurrent users, which defines request volume, allowing for a gradual increase in system stress in a regulated manner. Workloads range from light-load scenarios with fewer than 20 users to heavy-load settings with several hundred concurrently active virtual clients across the 15 training profiles (P1-P15). Read operations dominate in the low-intensity profiles, while write and update operations become more prevalent in the mid and high-intensity phases. This design reflects realistic workload evolution in production environments where traffic grows and data-modifying operations increase under elevated stress conditions. To evaluate model generalization under hitherto unforeseen combinations of traffic intensity and CRUD mixes, the three testing profiles (T1-T3) were held out. These profiles create a more difficult evaluation environment for anomaly detection and performance prediction tasks by introducing concurrency levels and operation ratios that are different from the training distribution. During the entire experiment, 10190 synchronized time-series data were collected by executing each workload profile for 600 seconds at a sampling frequency of 1 Hz. Transition periods between profiles were removed to preserve a continuous temporal structure. The Collector container aggregated database operations, application-level metrics (such as HTTP request rate and CRUD counts), and system-level resource information (such as CPU utilization, memory footprint, network I/O, and disk activity) into unified timestamped records during each run.

At specific workload intervals, the orchestrator introduces two classes of controlled anomalies in addition to standard operating conditions: brief CPU spikes in the API container and brief slowdowns in the PostgreSQL database. These anomalies are logged in real time and temporally aligned with the monitoring streams, enabling precise ground-truth labeling without modifying the workload execution logic. To provide a reproducible foundation for empirical analysis and subsequent machine learning experiments, this approach guarantees that every sample captures the instantaneous system state under the associated workload profile. Every timestamp is a combined snapshot of application-level performance and system-level resource usage. The synchronization of container measurements, database activity, and API replies enables consistent temporal modeling across layers [22, 23]. For portability and direct compatibility with machine learning workflows, all monitoring data is kept in comma-separated values (CSV) format. Each record contains 22 attributes, encompassing workload descriptors and resource metrics. The dataset structure is summarized in Table I. This unified data structure supports the time-series characterization of resource activity, anomaly detection, and correlation analysis. Fine-grained examination of system dynamics is made possible by a 1 Hz sampling resolution, which records brief workload spikes and performance anomalies. All experiments were conducted on a dedicated host. The system was an Apple MacBook Pro running macOS Ventura 14.5, equipped with an Apple M2 (8-core CPU), 16 GB of RAM, and a 512 GB SSD. A single docker-compose environment was used to deploy all of the containers with identical setups, and each workload profile was run separately. Through the collector module, system-level metrics were connected with application and database logs in almost real time.

TABLE I. SAMPLE STRUCTURE OF THE COLLECTED DATASET

Column	Description
datetime	Local timestamp sampled at 1 Hz; used to synchronize all monitoring streams
profile_id, phase	Workload identifier (P1-P15, T1-T3) and experiment phase (train/test)
users	Number of concurrent simulated users during the profile
rps	Request rate (requests per second) measured by locust
api_cpu_pct	CPU usage (%) of the API container, collected via docker stats and exposed for monitoring through Prometheus
api_mem_pct	Memory usage (%) of the API container, sourced from docker stats and monitored through Prometheus/cAdvisor
net_rx, net_tx	Network throughput (bytes/s) for receive and transmit directions
db_cpu_pct	CPU utilization (%) of the PostgreSQL database container
db_mem_pct	Memory usage (%) of the database container
db_connections	Current number of open PostgreSQL connections during the sampling interval
db_write_iops	Database write operations per second (IOPS), extracted from process-level disk stats
data_size	Total size (bytes) of stored records in postgreSQL, approximating dataset growth under write-heavy workloads
crud_create, crud_read, crud_update, crud_delete	Operation probability ratios defining workload composition
anomaly_flag	Binary indicator (0/1) specifying whether the sample lies inside a ground-truth anomaly interval
anomaly_type	Categorical label identifying the type of injected anomaly (cpu_spike_api, db_slowdown, or none for normal samples)
warmup_flag	Binary indicator (0/1) marking the warm-up period at the beginning of each workload profile where system metrics stabilize
warmup_state	Categorical warm-up status (warmup, steady), used to distinguish pre-stabilization intervals from steady-state system behavior

C. *Anomaly Injection Mechanism*

To embed realistic fault patterns into the monitoring dataset, anomalies were injected online during workload execution through an independent Orchestrator container [10,11]. This part updates an external state file that is later used by the Collector while simultaneously triggering controlled perturbations inside the database or API containers on a regular basis. Throughout the experiment, the anomaly creation process runs constantly and is divided into two classes: CPU spike at the API layer and database slowdown at the PostgreSQL layer. In the API container, a CPU spike is induced by launching a stress-ng task configured with two worker threads for a fixed duration. This results in a clean, isolated perturbation of the compute subsystem, causing a brief increase in CPU utilization without altering application logic or interfering with network and disk I/O [11]. On the other hand, a dual-stage load pattern is used inside the PostgreSQL container to cause the database delay. A minor connection storm increases the number of active connections and overloads session management by generating 4-6 concurrent sessions that run the blocking pg_sleep() instructions. In parallel, a controlled heavy-SQL loop continually runs a Cartesian expansion-based (generate_series(1,2000)) synthetic query on the items database, resulting in increased CPU load and decreased write throughput. To ensure smooth recovery, the heavy-SQL loop terminates slightly earlier than the sleep-based storm, after which the orchestrator waits for all blocking sessions to complete before the anomaly is considered finished. This prevents a backlog collapse that would otherwise distort tail-latency behavior [10]. Algorithm 1 outlines the exact procedures used to trigger both the CPU perturbation and the database slowdown.

Algorithm 1: Anomaly injection logic

```
Input:
    MIN_INTERVAL, MAX_INTERVAL
    DURATION = 13
    ANOMALIES = {cpu_spike_api, db_slowdown}
Procedure write_state(a):
    Write JSON {timestamp, anomaly=a} to STATE_FILE
Main Loop:
    write_state("none")
    while true:
        $\Delta t$ ← Uniform(MIN_INTERVAL, MAX_INTERVAL)
        sleep(Δt)
        a ← sample(ANOMALIES)
        write_state(a)
        if a = cpu_spike_api:
            run "stress-ng --cpu 2 --timeout DURATION" inside
API container
        if a = db_slowdown:
            in DB container:
                launch 4-6 pg_sleep() sessions in background
                execute heavy SQL loop until (DURATION − 2)
seconds
                wait for all pg_sleep() sessions to finish
        sleep(DURATION)
        write_state("none")
```

Both anomaly types are scheduled at random intervals, uniformly sampled between 100-150 seconds, and they run for a set duration of 13 seconds. The orchestrator adds an anomaly descriptor with the anomalous label and timestamp to a shared state file prior to activation; after the activity is done, the label is reset to zero. All anomaly periods are precisely in line with the exported time-series data because the Collector reads this file at a rate of 1 Hz. Thus, the mechanism guarantees deterministic labeling, reproducibility, and natural failure dynamics that are in line with actual microservice deployments.

III. Statistical Analysis and Findings

To describe system behavior under both typical and unusual operating situations, this part offers a statistical analysis of the gathered monitoring information. The analysis includes time-series inspection, distributional analysis, cross-metric correlation evaluation, and structure visualization based on dimensionality reduction. These statistical perspectives aid in illuminating the ways in which injected perturbations, resource consumption trends, and workload intensity appear in containerized system components. Fig. 2 shows the time-series behavior of RPS, database connections, DB write IOPS, and DB CPU utilization for each of the workload profiles (P1-P15, T1-T3). The db_slowdown anomalies are highlighted in red. The system shows a consistent pattern during abnormal intervals: write I/O activity collapses, throughput drops, queued queries generate a spike in DB connections, and CPU load dramatically increases due to heavy SQL operations and induced delays within PostgreSQL. These results demonstrate that db_slowdown generates a robust, consistent performance across various workload intensities.

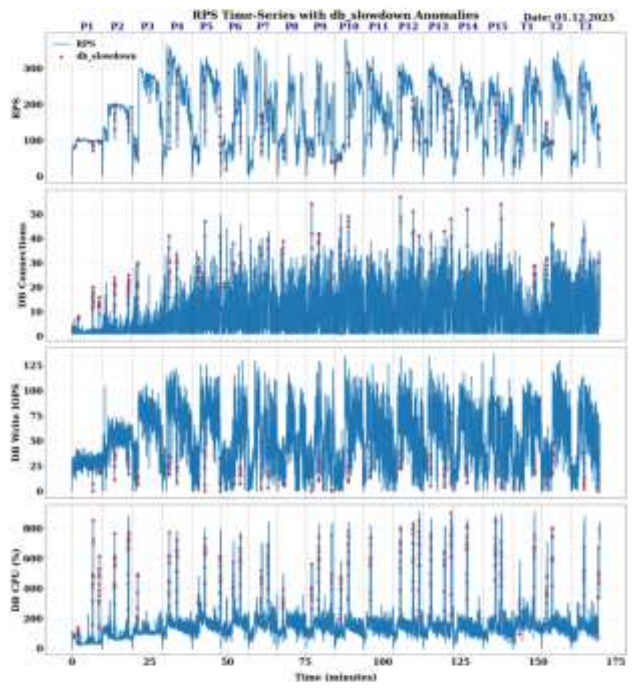

Fig. 2. Time-series behavior under db_slowdown anomalies

Fig. 3 shows the temporal dynamics of API CPU utilization, memory use, and request throughput under the cpu_spike_api anomaly for all workload profiles. While memory use shows smoother upward trends as a result of accumulated processing strain, the injected CPU spikes produce distinct, brief bursts in API CPU% (shown by red markers), frequently reaching saturation levels. These CPU spikes also correlate with localized throughput decreases, as the overburdened application server struggles to manage incoming requests. Sharp CPU peaks, slight memory shifts, and transient RPS degradation are the anomaly's crisp, highly localized hallmark.

Fig. 4 displays the empirical distributions of the main system-level variables gathered across all workload profiles. The metrics display various statistical behaviors: The multimodal and right-skewed forms of CPU-related measures (API and database CPU%) reflect transitions between low-load steady states and high-load stress phases. While database activity metrics like connections and write IOPS exhibit highly skewed, long-tailed patterns typical of bursty access to backend storage, throughput (RPS) has a broad, multi-peaked distribution that corresponds to various concurrency settings. Network traffic (net_rx and net_tx) exhibits similar right-skewed, heavy-tailed behavior, consistent with sporadic spikes during workload peaks. These distributions taken together indicate the importance of anomaly-aware modeling and machine learning approaches.

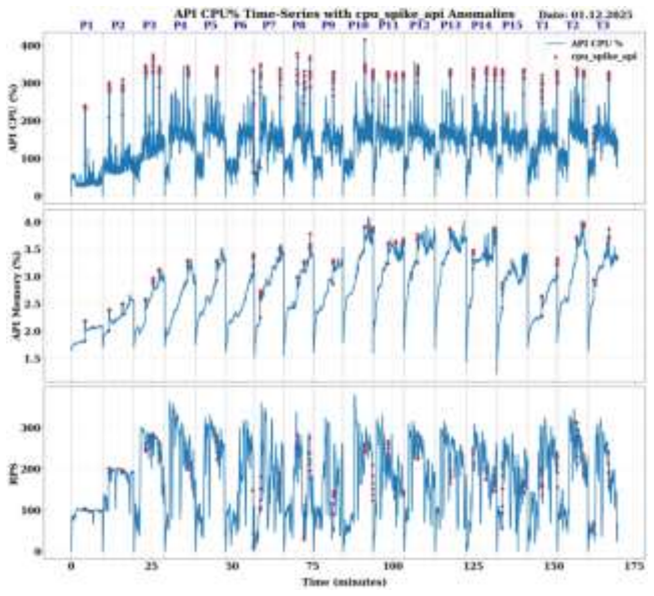

Fig. 3. API-level metrics under cpu_spike_api anomalies

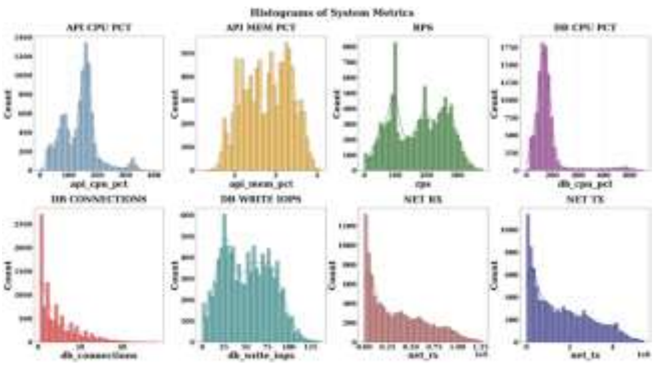

Fig. 4. Distributions of key system metrics

Fig. 5 summarizes the pairwise Pearson correlations among the system- and application-level monitoring variables. The matrix shows several clear relationships: While RPS is highly correlated with database write I/O activity, indicating the link between request volume and backend storage activities, API memory use shows strong positive correlations with network throughput. Additionally, moderate correlations show a common sensitivity to workload intensity between API CPU% and both RPS and db_write_iops. On the other hand, database CPU% exhibits weak or almost no correlations with the majority of indicators, indicating a decoupled execution profile controlled more by PostgreSQL's internal scheduling than by variations in external load. The overall pattern emphasizes the necessity for multivariate modeling techniques that can capture both correlated and orthogonal signals during anomaly detection.

Fig. 6 shows two complementary low-dimensional projections that reveal the geometric structure of normal and anomalous system states. The PCA projection shows a partial separation between normal and abnormal samples along the first two principal components. At higher PC2 values, db_slowdown events form a loose cluster, while cpu_spike_api points occupy a relatively dispersed region that overlaps the high-variance tail of normal behavior. The t-SNE embedding, on the other hand, shows more noticeable clustering, with cpu_spike_api points occupying several fragmented subregions that correspond to local shifts in resource-usage patterns and db_slowdown samples aggregating into compact and well-separated clusters. Collectively, these representations show that anomaly patterns appear as discrete nonlinear structures in the feature space.

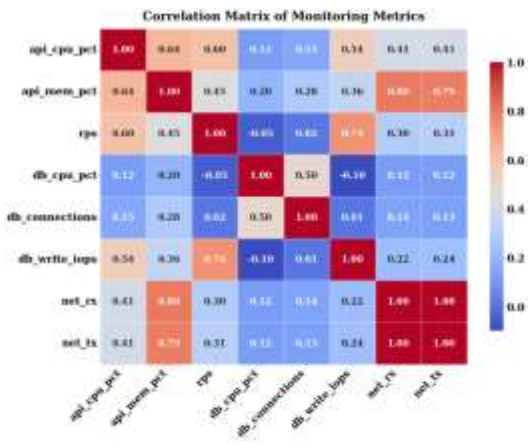

Fig. 5. Correlation matrix of monitoring metrics

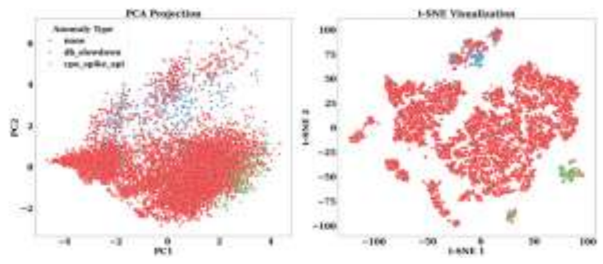

Fig. 6. PCA and t-SNE embeddings of the monitoring dataset

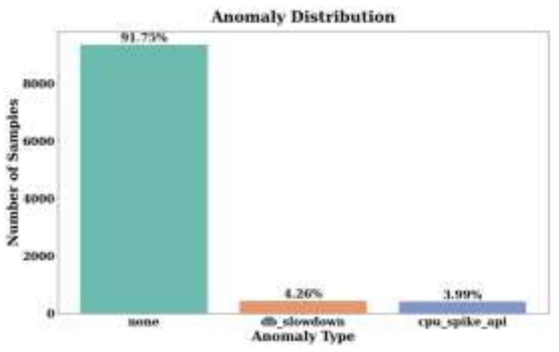

Fig. 7. Distribution of annotated anomalies across the dataset

Fig. 7 summarizes the relative frequencies of the two injected anomaly types in relation to typical system behavior. The dataset is highly unbalanced, with normal samples constituting over 90% of all observations, while db_slowdown and cpu_spike_api events account for approximately 4.26% and 3.99% of the total observations, respectively. The disparity emphasizes even more how crucial it is to have reliable detection methods that can spot a small number of aberrant patterns in largely regular operating data.

IV. Machine Learning Methods

A. *Problem Formulation and Evaluation Metrics*

The anomaly detection task in this study is formulated as a multiclass classification problem, where each timestamped system state is assigned to one of three classes: 0-normal, 1-cpu_spike_api, or 2-db_slowdown. Given a feature vector in Equation (1):

$$x_t = [f_1, f_2, ..., f_d] \tag{1}$$

The goal is to learn a mapping specified in Equation (2) that contains monitoring metrics at the API, database, and system levels at time t.

$$f : x_t \rightarrow y_t, y_t \in \{normal, cpu\_spike\_api, db\_slowdown\} \tag{2}$$

To ensure a comprehensive assessment of model performance, a set of supervised metrics commonly used in multiclass anomaly detection is employed:

- TP (True Positives): correctly detected anomalies.
- FP (False Positives): normal points incorrectly marked as anomalies.
- FN (False Negatives): anomalies missed by the model.
- TN (True Negatives): correctly identified normal points.

Accuracy is the fraction of correctly classified samples. Although intuitive, accuracy alone is insufficient under class imbalance [24].

Precision (macro-averaged), computed as the average precision over all classes C, treating each class equally regardless of frequency, is defined in Equation (3) [24,25].

$$\mathrm{Precision}_{macro} = \frac{1}{C}\sum_{i=1}^{C}\frac{TP_i}{TP_i + FP_i} \tag{3}$$

Recall (macro-averaged) is defined in Equation (4) and captures the ability to correctly identify anomaly samples [24,25].

$$\mathrm{Recall}_{macro} = \frac{1}{C}\sum_{i=1}^{C}\frac{TP_i}{TP_i + FN_i} \tag{4}$$

Macro-F1 is defined in Equation (5) as the most reliable metric under imbalance because it balances precision and recall per class and averages the results [24,25].

$$F1_{macro} = \frac{1}{C}\sum_{i=1}^{C} 2.\frac{\mathrm{Precision}_i . \mathrm{Recall}_i}{\mathrm{Precision}_i + \mathrm{Recall}_i} \tag{5}$$

Precision-Recall AUC (AUC-PR) quantifies the model's ranking performance across all classification thresholds and is generally more informative than ROC-AUC under heavy class imbalance. AUC-PR is defined in Equation (6) [25]. Where Precision(Recall) denotes the precision obtained at a given recall value.

$$AUC_{PR} = \int_0^1 \mathrm{Precision}(\mathrm{Recall})\, d(\mathrm{Recall}) \tag{6}$$

ROC-AUC is a threshold-independent metric that measures how well a model separates anomalous and normal states by integrating the True Positive Rate over the False Positive Rate and is defined in Equation (7) [25].

$$AUC_{ROC} = \int_0^1 TPR(FPR)\, d(FPR) \tag{7}$$

Where

$$TPR = \frac{TP}{TP + FN}, \quad FPR = \frac{FP}{FP + TN}$$

The confusion matrix is particularly useful for analyzing confusion between cpu_spike_api and db_slowdown events, as it summarizes class-wise errors and misclassifications. Together, these metrics enable robust comparison of supervised and unsupervised models and reveal their behavior under real-world monitoring conditions characterized by imbalance, multi-source signals, and heterogeneous anomaly patterns [24].

To ensure that both anomaly kinds are fairly evaluated and that the majority normal class does not dominate model performance, a balanced and trustworthy evaluation framework is provided by the use of macro-averaged metrics, ROC-AUC (one-vs-rest), and confusion matrices.

B. *Machine Learning Models and Experimental Setup*

As indicated in Table 2, a representative collection of machine learning models covering both supervised and unsupervised anomaly detection paradigms was chosen to evaluate the predictive utility of the generated dataset. This evaluation aims to investigate the responses of various learning principles, including linear classification, ensemble decision trees, gradient boosting, kernel-based outlier scoring, and reconstruction-based deviation detection, to the multivariate monitoring signals generated by the testbed, rather than optimizing any one model. To ensure a fair and reproducible comparison across algorithmic families, all models were trained and assessed using standardized features, identical train-test splits, and the shared set of metrics specified in Section 5.1.

TABLE II. Anomaly Detection Model Summary

Model Category	Algorithm	Description
Supervised	Logistic Regression	Linear multinomial classifier modeling class-conditional log-odds; baseline for separability analysis [26]
	Random Forest	Ensemble of decision trees using bootstrap aggregation; robust to noise and nonlinear interactions [27]
	XGBoost	Gradient-boosted tree model optimized via second-order gradient updates; strong performance in tabular anomaly detection [27]
	LightGBM	Histogram-based gradient boosting algorithm with leaf-wise tree growth; optimized for high-dimensional and large-scale data [27]
Unsupervised	Isolation Forest	Isolation-based anomaly scoring algorithm leveraging random tree partitioning of feature space [28]
	One-Class SVM	Kernel-based density boundary estimator that isolates abnormal instances in feature space [28]
	Local Outlier Factor (LOF)	Density-based anomaly detector identifying samples in low-density regions relative to local neighborhoods [28]

Model Category	Algorithm	Description
Supervised	Logistic Regression	Linear multinomial classifier modeling class-conditional log-odds; baseline for separability analysis [26]
	PCA Reconstruction Error	Linear subspace model where high reconstruction error indicates deviation from normal system behavior [28]

The experimental evaluation follows a unified and reproducible pipeline applied consistently across all machine learning models. All computations use the same train-test split, standardized preprocessing, and synchronized monitoring signals from the containerized testbed.

Algorithm 2: Anomaly detection pipeline

```
Input:
Require:
1. Training dataset $D_{train}$ testing dataset $D_{test}$
2. Monitoring features X, anomaly labels y (for supervised models)
3. Selected ML models $M = [M_1, ..., M_k]$
4. Standardization function Scaler(·)
5. Evaluation metrics: Accuracy, Precision, Recall, Macro-F1, AUC-PR, ROC-AUC
6. For unsupervised models: anomaly-score function S(·), threshold selection rule $\tau$
Initialize:
7. Load raw time-series data collected from API, DB, and system telemetry
8. Remove transient container restart intervals
9. Align all monitoring signals at 1 Hz sampling resolution
For each dataset
1. Preprocessing:
1.1 Construct feature matrix X from synchronized metrics
1.2 Encode anomaly labels y(supervised) or set y ← None (unsupervised)
1.3 Apply scaling:
        $X'_{train} = Scaler.fit_{transform(X_{train})}$, $X'_{test} = Scaler.fit_{transform(X_{test})}$
2. Training:
For each model $M_k \in M$:
2.1 If supervised:
        $M_k \leftarrow M_k.fit(X'_{train}, Y_{train})$
2.2 If unsupervised:
        $M_k \leftarrow M_k.fit(X'_{train})$
3. Inference:
3.1 Supervised:
        $\hat{y} = M_k.predict(X'_{test})$, $\hat{p} = M_k.predict\_proba(X'_{test})$
3.2 Unsupervised:
        $s = S(M_k, X'_{test})$, $\hat{y} = 1[s > \tau]$
4. Evaluation:
4.1 Compute Accuracy, Precision, Recall, Macro-F1
4.2 Compute AUC-PR, ROC-AUC (one-vs-rest for supervised, binary for unsupervised)
4.3 Construct confusion matrix
4.4 Store all metrics into result table
Return:
10. Trained models $M$
11. Evaluation summary tables
12. Confusion matrices
13. ROC curves and PR curves
14. SHAP feature-importance explanations (for tree-based models)
```

The pipeline works with multivariate time-series data, finds both labeled and unlabeled anomalies, scales features, trains both supervised and unsupervised models in the same way, and tests them with the same set of metrics. This controlled setup ensures a fair comparison and a realistic evaluation of model accuracy and anomaly sensitivity. Algorithm 2 fully formalizes the evaluation pipeline.

C. Results

Table 3 summarizes the performance of all eight anomaly detection models by contrasting supervised and unsupervised methods using a variety of evaluation parameters. The supervised classifiers (logistic regression, random forest, XGBoost, and LightGBM) continuously perform well, achieving accuracies above 0.96 and macro-F1 scores between 0.82 and 0.85, while exhibiting robust generalization across the three operating stages of the system. Logistic regression has the best ROC-AUC (0.972), random forest has the highest macro-F1 (0.853), and gradient-boosting models offer competitive, well-balanced precision-recall behavior (AUC-PR ~ 0.88). In contrast, because there is no labeled supervision, the unsupervised models exhibit noticeably poorer discriminative capacity. In this group, Isolation Forest performs best (AUC-PR = 0.323; ROC-AUC = 0.879), while ROC-AUC values are less than 0.77 for One-Class SVM, LOF, and PCA reconstruction error. All things considered, the findings highlight the distinct benefit of supervised learning in this multi-modal, containerized setting and provide a solid foundation for the ensuing ROC, precision-recall, and confusion-matrix analyses.

TABLE III. COMPARISON OF METRICS

Model	Accu-racy	Precision	Recall	macro-F1	AUC-PR	ROC-AUC
Logistic regression	0.968	0.911	0.780	0.834	0.886	0.972
Random forest	0.970	0.922	0.799	0.853	0.883	0.952
XGBoost	0.964	0.882	0.779	0.821	0.877	0.959
LightGBM	0.969	0.906	0.795	0.843	0.887	0.965
Isolation forest	-	-	-	-	0.323	0.879
One-class SVM	-	-	-	-	0.187	0.768
LOF	-	-	-	-	0.087	0.267
PCA reconstruction error	-	-	-	-	0.131	0.642

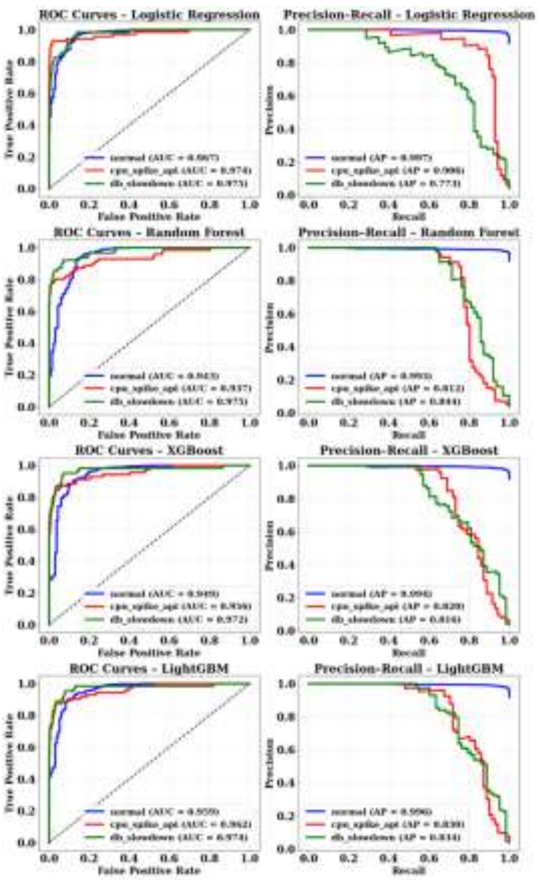

Fig. 8. ROC and precision-recall curves on supervised models

Figs. 8-9 present the ROC and Precision-Recall (PR) curves for all evaluated models, illustrating differences in separability and anomaly-detection reliability across supervised and unsupervised approaches. The supervised models (logistic regression, random forest, XGBoost, and LightGBM) have steep ROC curves and high AUC values for all types of anomalies. This means that they can tell the difference between normal and anomalous states. Their PR curves likewise maintain high precision at varying recall levels, especially for the dominant none class, reflecting robust performance under class imbalance. In contrast, the unsupervised methods demonstrate noticeably weaker separation. While Isolation Forest achieves a reasonably convex ROC curve, its PR curve reveals difficulty maintaining precision at higher recall levels. Limited discriminatory capacity is demonstrated by One-Class SVM and PCA reconstruction error, while LOF exhibits poor performance on both ROC and PR metrics, underscoring its susceptibility to high-dimensional noise and nonstationary workload patterns. Overall, these visual results reinforce the quantitative findings in Table 3 and confirm the superior reliability of supervised models for anomaly detection in this containerized monitoring dataset.

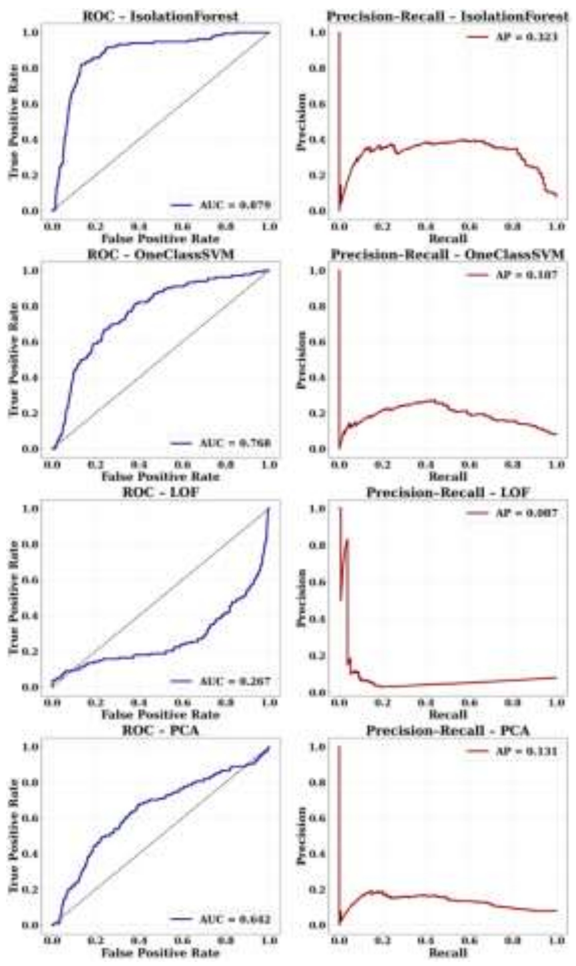

Fig. 9. ROC and precision-recall curves on unsupervised models

Fig. 10 compiles the classification behavior of all supervised and unsupervised models. With only slight confusion in borderline high-load areas, the supervised approaches consistently demonstrate high accuracy, correctly identifying the majority of normal data and consistently distinguishing between the two anomaly kinds. The unsupervised methods, on the other hand, perform significantly worse: While one-class SVM, LOF, and PCA reconstruction errors show poor discrimination, with decision boundaries unable to capture the structure of the monitoring data, isolation forests offer the most stable anomaly separation but still generate a large number of false positives. These findings support the notion that supervised and unsupervised techniques perform significantly differently on this dataset.

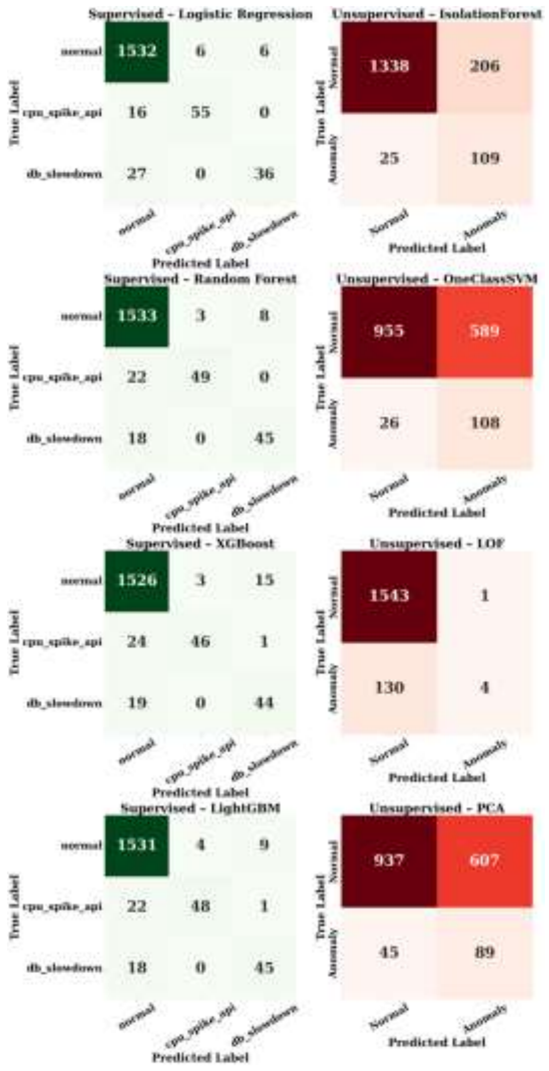

Fig. 10. Confusion matrices

Fig. 11 displays the SHAP-based feature importance profiles for Random Forest, XGBoost, and LightGBM, along with a combined global ranking. Both API and database-level load signals have a significant impact on model decisions, as evidenced by the fact that api_cpu_pct consistently emerges as the major predictor of anomalies across all three models, followed by db_write_iops, db_mem_pct, and db_cpu_pct. Although of lesser importance, request-level indicators like rps and db_connections, as well as network throughput measures like net_rx and net_tx, also rank among the top contributors. These patterns are further supported by the aggregated ranking, which shows a consistent cross-model agreement on the metrics that are most susceptible to aberrant system conditions. All of these findings show that the learned models are highly dependent on CPU and I/O stress indicators, which is consistent with the injected anomalies' operational features.

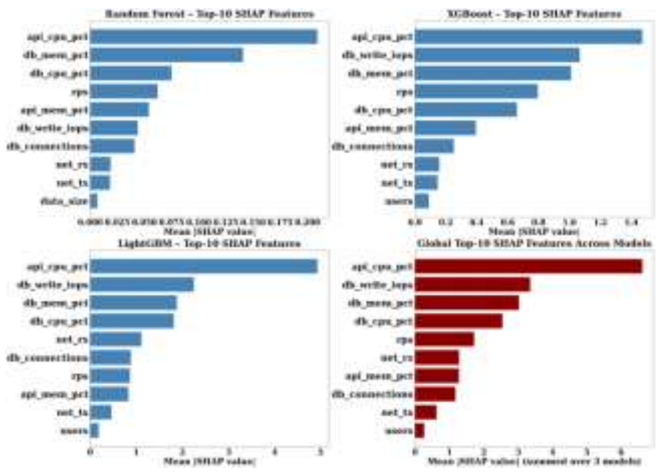

Fig. 11. SHAP feature importance across supervised tree-based models

Overall, the machine learning evaluation demonstrates that when it comes to identifying abnormal behavior in the containerized API-database system, supervised models consistently perform better than unsupervised methods. While unsupervised detectors suffer from the intrinsic challenge of modeling infrequent anomalies without labels, supervised classifiers produce solid ROC-AUC/PR-AUC performance, excellent accuracy, and outstanding macro-F1 scores. Stable feature-importance trends across models are further highlighted by SHAP studies, which show that CPU load and I/O measurements are the most important predictors of aberrant states. All of these findings support the usefulness of the produced dataset as well as the efficacy of learning-based techniques for anomaly identification in controlled microservice environments.

V. CONCLUSION

This work fully containerized the experimental testbed for creating infrastructure monitoring datasets enhanced with realistic anomaly patterns. Through the integration of controlled load creation, multi-layer telemetry collecting, and managed anomaly injection, the platform makes it possible to observe application database behavior under various operating scenarios in a repeatable and detailed manner. High temporal resolution and reliable synchronization across workload, application, and resource metrics are provided by the resulting dataset, which includes 18 workload profiles and two system-level anomaly classes. The two anomaly types, cpu_spike_api and db_slowdown, exhibit unique and comprehensible fingerprints in both application-level performance and system-level resource use, according to statistical analysis. Additionally, machine learning experiments indicated that classical supervised models can detect similar patterns, with Random Forest and LightGBM exhibiting the best overall performance. Although more difficult in this context, unsupervised approaches nevertheless demonstrated significant separability in the feature space and highlight opportunities for improvement through more expressive temporal models. Overall, the proposed testbed offers a transparent and consistent environment for evaluating anomaly detection techniques in modern containerized environments. It provides a dataset that is accessible to the public as well as a methodological framework that can facilitate more investigation into the relationship between workload behavior, resource usage, and system faults.

Future research could enhance the testbed by incorporating additional anomaly types, such as disk saturation, network congestion, or memory leaks, along with autoscaling algorithms to evaluate detection amidst dynamic resource fluctuations. Reinforcement-learning techniques can enhance the anomaly generator, enabling it to produce fault patterns that are more diverse and adaptable. Additionally, a viable approach to enhance sequential anomaly identification in multivariate monitoring data is to use deep temporal models like transformer-based topologies, TCN, or LSTM.

Data Availability Statement: All data and programs are available https://doi.org/10.17632/drkg8r6tfr.1

REFERENCES

[1] Z. Zhong, Q. Fan, J. Zhang, M. Ma, S. Zhang, Y. Sun, Q. Lin, Y. Zhang, and D. Pei, "A survey of time series anomaly detection methods in the AIOps domain," arXiv preprint arXiv:2308.00393, 2023.

[2] F. Wang, Y. Jiang, R. Zhang, A. Wei, J. Xie, and X. Pang, "A survey of deep anomaly detection in multivariate time series: taxonomy, applications, and directions," Sensors, vol. 25, no. 1, Art. no. 190, 2025.

[3] A. Garg, W. Zhang, J. Samaran, S. Ramasamy, and C.-S. Foo, "An evaluation of anomaly detection and diagnosis in multivariate time series," IEEE Trans. Neural Netw. Learn. Syst., 2021.

[4] D. Abshari and M. Sridhar, "A survey of anomaly detection in cyber-physical systems," arXiv preprint arXiv:2502.13256, 2025.

[5] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using LSTMs and nonparametric dynamic thresholding," in Proc. 24th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2018, pp. 387–395.

[6] M. Llugiqi and R. Mayer, "An empirical analysis of synthetic-data-based anomaly detection," in Machine Learning and Knowledge Extraction, Lecture Notes in Computer Science, 2022.

[7] Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-world challenges and research innovations," in Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Companion Proc. (ICSE-Companion), May 2019.

[8] V. Mahavaishnavi, R. Saminathan, and R. Prithviraj, "Container security intelligence: Leveraging machine learning for anomaly detection in containerized applications," Tuijin Jishu/Journal of Propulsion Technology, vol. 44, no. 3, 2023.

[9] M. Raeiszadeh, A. Ebrahimzadeh, R. H. Glitho, J. Eker, and R. A. F. Mini, "Asynchronous real-time federated learning for anomaly detection in microservice cloud applications," IEEE Trans. Mach. Learn. Commun. Netw., vol. 3, pp. 176–194, 2025.

[10] T. Higgins, D. N. Jha, and R. Ranjan, "Swarm storm: An automated chaos tool for Docker Swarm applications," in Proc. 33rd Int. Symp. High-Performance Parallel and Distributed Computing (HPDC), 2024, pp. 367–369.

[11] S. Sondhi, S. Saad, K. Shi, M. Mamun, and I. Traore, "Chaos engineering for understanding consensus algorithms performance in permissioned blockchains," arXiv preprint arXiv:2108.08441, 2021.

[12] M. Allam, N. Boujnah, N. E. O'Connor, and M. Liu, "Synthetic time series for anomaly detection in cloud microservices," arXiv preprint arXiv:2408.00006, 2024.

[13] M. S. Islam, M. S. Rakha, W. Pourmajidi, J. Sivaloganathan, J. Steinbacher, and A. Miranskyy, "Anomaly detection in large-scale cloud systems: An industry case and dataset," in Proc. IEEE/ACM 47th Int. Conf. Software Engineering: Software Engineering in Practice, 2025.

[14] S. V. Kochergin, S. V. Artemova, A. A. Bakaev, E. S. Mityakov, Z. G. Vegera, and E. A. Maksimova, "Cybersecurity of smart grids: Comparison of machine learning approaches training for anomaly detection," Russian Technological Journal, vol. 12, no. 6, pp. 7–19, 2024.

[15] D. Kohyarnejadfard, D. Aloise, S. V. Azhari, and M. R. Dagenais, "Anomaly detection in microservice environments using distributed tracing data analysis and NLP," Journal of Cloud Computing, 2022.

[16] E. Hitimana, G. Bajpai, R. Musabe, L. Sibomana, and K. Jayavel, "Containerized architecture performance analysis for IoT framework based on enhanced fire prevention case study: Rwanda," Sensors, vol. 22, no. 17, Art. no. 6462, 2022.

[17] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," arXiv preprint

[18] S. Magomedov, D. Ilin, and E. Nikulchev, "Resource analysis of the log files storage based on simulation models in a virtual environment," Applied Sciences, vol. 11, no. 11, Art. no. 4718, 2021.

[19] Q. Wang, S. Kar, P. Mishra, C. Linduff, R. Izard, K. Anjam, G. Barrineau, J. Zulfiqar, and K.-C. Wang, "Container resource allocation versus performance of data-intensive applications on different cloud servers," arXiv preprint arXiv:2311.07818, 2023.

[20] G. Turin, A. Borgarelli, S. Donetti, F. Damiani, E. B. Johnsen, and S. L. Tapia Tarifa, "Predicting resource consumption of Kubernetes container systems using resource models," Journal of Systems and Software, vol. 203, Art. no. 111750, 2023.

[21] T. Rak, "Performance evaluation of an API stock exchange web system on cloud Docker containers," Applied Sciences, vol. 13, no. 17, Art. no. 9896, 2023.

[22] J. Wang, Y. Li, C. Jiang, and S. Guo, "Container performance prediction: Challenges and solutions," in Communications and Networking, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2021.

[23] S. Chen, "Locust swarms: A new multi-optima search technique," in Proc. IEEE Congress on Evolutionary Computation (CEC), 2009.

[24] P. Christen, D. J. Hand, and N. Kirielle, "A review of the F-measure: Its history, properties, criticism, and alternatives," ACM Computing Surveys, vol. 56, no. 3, Art. no. 73, pp. 1–24, 2023.

[25] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in Proc. 23rd Int. Conf. Machine Learning (ICML), 2006, pp. 233–240.

[26] M. K. Chung, "Introduction to logistic regression," arXiv preprint arXiv:2008.13567, 2020.

[27] Y. M. Indah, R. Aristawidya, A. Fitrianto, and E. Erfiani, "Comparison of random forest, XGBoost, and LightGBM methods for the human development index classification," Jambura Journal of Mathematics, vol. 7, no. 1, pp. 14–18, 2025.

[28] R. P. Nathan, N. Nikolaou, and O. Lahav, "Finding Pegasus: Enhancing unsupervised anomaly detection in high-dimensional data using a manifold-based approach," arXiv preprint arXiv:2502.04310, 2025.

Minh Hieu Nguyen, master's student at the department of Data processing digital technologies, MIREA – Russian technological university (e-mail: nguen.m2@edu.mirea.ru, https://orcid.org/0009-0001-7782-0563).

Dmitry Ilin, Ph.D. (Tech.), associate professor at the department of Data processing digital technologies, MIREA – Russian technological university (e-mail: i@dmitryilin.com, https://orcid.org/0000-0002-0241-2733).