I. J. Engineering and Manufacturing, 2026, 3, 218-237
Published Online on June 8, 2026 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijem.2026.03.13

Feature Selection Methods for Intelligent Software Performance Monitoring Based on Machine Learning

Liubov Oleshchenko*
Computer Systems Software Department, Faculty of Program Systems and Applied Mathematics, Igor Sikorsky Kyiv Polytechnic Institute, Kyiv, 03056, Ukraine
E-mail: oleshchenkoliubov@gmail.com
ORCID ID: https://orcid.org/0000-0001-9908-7422
*Corresponding author

Zhengbing Hu
School of Computer Science, Hubei University of Technology, Wuhan, 430079, China
E-mail: drzbhu@gmail.com
ORCID ID: https://orcid.org/0000-0002-6140-3351

Andrii Dychka
Computer Systems Software Department, Faculty of Program Systems and Applied Mathematics, Igor Sikorsky Kyiv Polytechnic Institute, Kyiv, 03056, Ukraine
E-mail: andriydychka@gmail.com
ORCID ID: https://orcid.org/0000-0003-0578-2788

Received: 27 March, 2026; Revised: 16 April, 2026; Accepted: 26 April, 2026; Published: 08 June, 2026

Abstract: This paper presents a set of feature selection methods for intelligent software performance monitoring based on machine learning models, with a focus on improving interpretability, scalability, and adaptability in high-dimensional telemetry analysis. The research addresses limitations of traditional statistical and rule-based approaches, which are often unable to capture nonlinear dependencies and dynamic interactions in modern distributed architectures. A unified methodology is proposed that integrates several complementary techniques for adaptive feature selection in intelligent monitoring systems. These include a topology-aware method based on graph neural networks for modeling structural dependencies in microservice architectures, a correlation-driven approach for reducing feature redundancy, a multifactor fusion method combining statistical significance, temporal stability, and predictive contribution, a cost-efficient strategy for serverless environments, and a context-aware reinforcement learning approach for dynamic feature adaptation. The proposed methods are evaluated on a large-scale dataset exceeding 3.5 TB, collected from 42 real-world applications representing monolithic, microservice, cloud-native, and serverless architectures. The results show an average reduction in feature dimensionality of 37%, while maintaining over 95% predictive accuracy across multiple models. Additional improvements include, on average, a 21% increase in dependency modeling accuracy, an 18% gain in feature relevance estimation, a 26% reduction in feature instability under dynamic workloads, and up to 42% cost reduction in serverless environments, as observed across repeated experiments under controlled workload variability and consistent evaluation settings. While the results demonstrate the effectiveness of adaptive feature selection, further validation in diverse real-world conditions is required to confirm the generalizability of the proposed framework.

Index Terms: Feature Selection Methods, Machine Learning, Web Applications, Telemetry Data, Adaptive Models, Cloud Systems, Microservices, Intelligent Software Monitoring

1. Introduction

The rapid growth of modern distributed software systems generates large volumes of high-dimensional telemetry data, creating significant challenges for efficient performance monitoring and analysis. The increasing complexity of web applications, driven by microservice architectures, cloud-native platforms, and serverless computing, leads to highly dynamic execution environments characterized by variable workloads, heterogeneous components, and evolving

This work is open access and licensed under the Creative Commons CC BY 4.0 License.

system dependencies. In such conditions, traditional monitoring approaches based on static rules, predefined thresholds, and aggregated indicators become insufficient, resulting in reduced analytical accuracy and limited adaptability.

The foundations of feature selection in machine learning were established in classical statistical and information-theoretic approaches, which focus on identifying relevant variables while reducing data dimensionality. Methods such as LASSO regularization, mutual information analysis, and principal component analysis have been widely applied for reducing redundancy and improving model interpretability. These approaches provide a basis for modern feature selection techniques; however, they are typically designed for static datasets and do not account for dynamic system behavior or structural dependencies in distributed environments.

Recent research demonstrates a transition toward adaptive and hybrid feature selection methods that combine statistical metrics, optimization algorithms, and machine learning models. Such approaches improve model stability, reduce variance, and enhance predictive performance, particularly in high-dimensional and heterogeneous data scenarios. In parallel, advances in telemetry analysis and distributed tracing enable more detailed observation of system behavior, providing new opportunities for identifying structurally significant features. Graph-based methods, including graph neural networks, further extend this capability by modeling dependencies between system components and improving feature relevance estimation in complex architectures. The integration of machine learning into software performance monitoring has enabled automated anomaly detection, predictive analysis, and intelligent diagnostics. However, the effectiveness of these approaches strongly depends on the quality and relevance of selected features. High-dimensional telemetry data often contain redundant, correlated, and noisy variables, which increase computational overhead and reduce model robustness. Existing feature selection techniques demonstrate limitations in dynamic environments, as they often fail to provide stable and interpretable feature subsets under changing workloads and system configurations.

Modern monitoring platforms still lack unified mechanisms for adaptive feature selection that simultaneously consider statistical properties of data, structural characteristics of software systems, and operational context. As a result, feature selection is often performed using isolated methods that do not fully exploit the available information. This creates the need for a comprehensive approach capable of dynamically selecting relevant features based on both data-driven and system-level factors. The goal of this article is to develop and experimentally validate adaptive methods for selecting key features in web application performance analysis using machine learning models. The proposed approach aims to ensure high predictive accuracy, scalability, and computational efficiency under dynamic operating conditions.

1.1. Problem Statement

Given a high-dimensional telemetry dataset generated by a distributed software system operating under dynamic workload conditions, the task is to determine an optimal subset of features that maximizes predictive performance while minimizing redundancy and computational cost. The inputs to the system include telemetry data streams containing performance metrics, logs, and traces, as well as contextual information such as system architecture type, workload characteristics, and resource constraints. Additional inputs include statistical properties of features and structural relationships between system components. The expected outputs are a selected feature subset F_k, reduced feature dimensionality, and improved model performance metrics, including predictive accuracy, stability, and computational efficiency. The system must operate under dynamic conditions, ensuring adaptability to workload changes, reproducibility of feature selection decisions, and bounded computational overhead. Additionally, the framework should maintain interpretability of selected features and support integration with machine learning models for performance analysis and anomaly detection.

1.2. Motivation

The growing complexity of modern software systems highlights the limitations of traditional feature selection approaches, which are typically static and do not account for dynamic system behavior. In distributed environments, inefficient feature selection can lead to excessive data processing, reduced model accuracy, and unstable analytical results.

Recent research emphasizes the importance of adaptive, context-aware approaches that combine data-driven analysis with system-level considerations. However, existing methods often focus on isolated aspects, such as statistical relevance or model-based importance, without integrating structural dependencies and operational constraints.

This work is motivated by the need to bridge the gap between theoretical feature selection methods and practical monitoring systems. The proposed framework addresses this challenge by integrating statistical analysis, structural modeling, and adaptive decision-making into a unified approach. The main idea of the proposed methods is to model feature selection as a multi-criteria decision problem, where the system continuously evaluates trade-offs between predictive accuracy, feature dimensionality, computational cost, and system context. The framework dynamically selects feature subsets based on statistical relevance, structural importance, and workload conditions, enabling efficient and scalable performance monitoring in complex distributed environments.

The goal of the article is to develop and experimentally validate adaptive methods for selecting key features for web application performance analysis based on machine learning models, ensuring high accuracy, scalability, and efficiency under dynamic operating conditions.

2. Related Work

Modern research in intelligent software performance monitoring increasingly focuses on machine learning techniques for processing high-dimensional telemetry data and performing adaptive feature selection. Machine learning-based methods for performance evaluation have demonstrated significant improvements compared to traditional approaches, particularly when combined with optimization strategies, enabling more accurate modeling of complex system behavior [1]. Within this context, feature selection plays a central role in reducing dimensionality while preserving relevant information. Comparative studies show that hybrid feature selection methods improve model stability and reduce variability, making them suitable for large-scale telemetry data analysis [2]. At the same time, the relationship between feature importance metrics and predictive accuracy remains complex, as improper metric selection can lead to performance degradation [3]. The development of dedicated evaluation metrics for feature selection algorithms further enables systematic comparison in terms of efficiency, robustness, and stability [4].Classical feature selection approaches, including LASSO and mutual information-based methods, remain widely used due to their computational efficiency and interpretability. These methods allow substantial reduction of feature dimensionality while maintaining acceptable predictive performance [5], [6]. More recent approaches combine statistical techniques with optimization algorithms, leading to improved accuracy and stronger dimensionality reduction, particularly in heterogeneous and high-dimensional data environments [7]. In distributed software systems, particularly microservice-based architectures, telemetry analysis is closely related to observability and tracing mechanisms. Industrial studies demonstrate that distributed tracing significantly improves system diagnostics and performance analysis by providing insights into inter-service interactions [8]. Advanced tracing pipelines and locality-aware analysis further enhance the identification of dependencies between system components [9]. These capabilities are highly relevant for feature selection, as they enable the identification of structurally significant features derived from system interactions.Anomaly detection and root cause analysis in distributed systems also depend on effective feature selection. Recent methods based on telemetry traces and profiling data achieve high detection accuracy while improving diagnostic efficiency [10], [11]. In this context, modeling dependencies between system components becomes increasingly important. Graph-based approaches, including graph neural networks, provide effective mechanisms for capturing structural relationships and improving feature relevance estimation in complex systems [12], [13]. Industrial studies further indicate that continuous performance optimization and monitoring can lead to measurable improvements in system efficiency, typically in the range of 10–18% [14].

Recent studies emphasize the increasing importance of intelligent monitoring and observability in microservice-based systems. F.Gomes et al. (2025) provide a comprehensive mapping of observability approaches, highlighting challenges related to scalability, data heterogeneity, and the lack of adaptive analysis mechanisms. Y. Hammad et al. (2025) investigate the performance overhead introduced by instrumentation in containerized environments, demonstrating the trade-off between monitoring accuracy and system efficiency.

Overall, current research trends demonstrate a shift from isolated feature selection techniques toward integrated, adaptive, and context-aware approaches. These approaches combine statistical methods, machine learning models, telemetry analysis, and structural modeling to address the complexity of modern distributed systems. However, the development of a unified feature selection framework that simultaneously considers system architecture, dynamic workload conditions, and computational cost remains an open problem, which defines the main contribution of this study.

3. Proposed Feature Selection Methods for Intelligent Software Performance Monitoring

3.1. Topology-oriented feature selection method based on graph neural network model

Classical methods for selecting performance features are focused on statistical dependencies between individual indicators, but they do not take into account structural interactions between services. Based on the conclusions obtained in the first section, to develop a method for classifying web applications by their deployment architecture, *telemetric* and *structural* indicators of web applications were identified for further research.

Telemetry metrics reflect the behavior of a web application when it is running in a real environment. Telemetry characteristics include: latency – the time that elapses from the moment the user sends a request until receiving a response from the server; throughput – the number of requests or operations that the system is able to process per unit of time; response time – the average or maximum execution time of individual operations, for example, page generation or API response; load distribution – an indicator of the evenness of resource use between servers or system components. These parameters allow assess how the system responds to load changes, which nodes are “bottlenecks,” and what its performance profile is. Based on the web application architectures analyzed in the first section, their main *structural performance indicators* were identified - parameters that describe the architectural structure of the web application: the number and type of interconnected components, the method of their communication, dependencies between services, types of interfaces, deployment schemes: number of services in the system; interactions between components; the number of direct dependencies between components (level of connectivity); a way of managing the state. In microservice architectures, each component affects others through network calls, message queues, load balancers, and API gateways.

Topology refers to a structural model of software architecture, presented in the form of a graph, where nodes are services and edges are calls or dependencies between them.

TAM-GNN (Topology-Aware Metrics selection via Graph Neural Networks) method formalizes the structure of a web application as a directed graph and selects features taking into account structural influences between web application components. This ensures that statistical dependencies between service performance indicators are taken into account, key nodes and features that have the greatest impact on global degradations are selected, and the set of observed features is dynamically updated in accordance with changes in the system topology.

The purpose of the proposed method is to automatically classify the architectural type of a web application based on a combination of telemetric indicators and structural relationships between components, which is important for distributed systems. The use of graph neural networks GNN (Graph Neural Networks) is due to the fact that they reflect the topological structure of systems, where each service affects others through dependencies in queries. Unlike classical neural network models based on tabular or sequential data (RNN, CNN), GNNs take into account the relationships between components through aggregation with neighbors and allow us to assess the influence of nodes in the context of the architecture, and not in isolation, support dynamic graphs, i.e. they can adapt to changes in topology when updating microservices. The use of GNN allows to take into account the topology of the architecture, i.e. how exactly the components are interconnected; identify interaction patterns that affect performance; to form embeddings that reflect the architectural structure in a numerical form suitable for combining with telemetric features.

The proposed adaptation of the TAM-GNN model provides a mechanism for topology-oriented feature selection that minimizes the amount of monitoring data without losing analytical accuracy.

In the context of web applications, the architecture of a web application can be represented as a graph $G = (V, E)$, where V is a set of nodes (services or modules), and E is a set of edges (connections or calls between them). Each node can have its own attributes – performance indicators, configuration parameters, type of service.

The edge $e = (u, v)$ shows the direction of the call: service u calls or sends a request to service v, this corresponds to the typical behavior of microservices, where one component initiates a request to another. Vertices are services or containers (e.g. AuthService, API Gateway, OrderService, Database) denoted as $V = \{v_1, v_2, v_3, ...\}$. Local metrics are stored for each vertex, for example:

$$x_v = [\ avg\ CPU, peak\ memory, avg\ latency, error\ rate, requests\ ] \quad (1)$$

Edges are connections between services that represent calls (*HTTP/gRPC, message queues*, etc.): $e = (u, v)$, $x_e =$ [*latency, RPS, error rate, retries*], they have a direction and attributes that describe the interaction between components. This model allows to build a call graph from OpenTelemetry traces, analyze service performance and interdependencies, identify "bottlenecks" or services with high error rates or latency, and use the graph as a basis for anomaly detection, service clustering, or web application topology visualization.

To implement the method, it was proposed to build a hybrid pipeline, the main stages of which are described below.

Stage 1. Collection and normalization of telemetry, logs, and metadata.
Stage 2. Construction of a graph $G=(V,E)$, representing components/hosts/services as vertices, calls/channels as edges, with corresponding attributes on vertices and edges.
Stage 3. Application of the GNN model to obtain structural vector representations of graph vertices.
Stage 4. Vectorization of time aggregated indicators.
Stage 5. Combining vector representations and presenting them to the ensemble classifier (XGBoost over GNN-vector representations and combining with tabular features).
Stage 6. Training the model on a set of marked examples of web application architectures.

For each application *a,* we have a set of time indicators $M_a(t)$ that vary in time and describe the operation of the application *a*; *CPU* – CPU load (percentage of usage or absolute values of cycles/sec); *memory* – RAM usage in MB/GB; *response time* – response time for processing a request in milliseconds; *throughput* – bandwidth, number of requests per second, RPS (*requests per second*). Each indicator is a time series $m(t)$, that is, values that arrive or are measured at points in time. Raw time series data is very large and difficult to process in batch mode. Statistical aggregates preserve key information (central tendency, tails, variability, trends), reduce the size of the data, and make the models more stable. The p95/p99 percentiles are needed for performance analysis, as they show extreme delays that are critical for the user experience. Instead of working with raw time series (a large number of points in time), a compact statistical feature vector of dimension p is constructed for each web application. Each component of this vector is a summary statistic or characteristic of the time series. Let R^p be a vector of p real numbers. The basic statistical characteristics (mean, median, standard deviation σ, and percentiles) are represented as an aggregated vector $m_a \in R^p$. For our problem, aggregated statistical characteristics were chosen that are used to describe time series of web application performance indicators. Aggregation is performed in fixed time windows (1-minute or 5-minute).

For a window T using all dimensions $x_1, ..., x_n$ are calculated, $\bar{x}$, σ, where $\bar{x}$ *is* the arithmetic mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (2)$$

where x_i is the value of the indicator at each point in time, n is number of measurements in the aggregation window. Sliding windows are used for streaming processing. To detect a trend in the time series, the slope of the linear regression approximation is calculated. Standard deviation σ *is* a measure of dispersion:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})} \tag{3}$$

Percentiles are values below which a certain percentage of observations lie, p95 (95th percentile) is the number q such that 95% of the values $\leq q$. Formally:

$$q_p = \inf\{x : F(x) \geq p\} \tag{4}$$

where F is empirical distribution function.

These statistics provide insight into the central tendency, dispersion, and "tails" of the distribution of the indicator as input features for a machine learning model. The sources for constructing the graph are traces from OpenTelemetry, call logs in spans format, and recorded HTTP/gRPC calls between services. Each vertex $v \in V_a$ is a service/container/process/host, with feature vector x_v (aggregated indicators for a node). Each node has its own set of local characteristics: average CPU for this node over the time window; peak memory; average query processing time locally; fraction of errors associated with this node; number of queries processed. These characteristics are also aggregate statistics, similar to those described above for the global vector m_a, but localized for each node. Each edge $e = (u, v) \in E_a$ is a connection or call with characteristics x_e (average delay between services, *rps, error_rate*).

An edge also has its own characteristics that describe the interaction between the two components: the average latency of calls from service u to service v; RPS (*requests per second*) between u and v; *error_rate* for calls u to v; frequency of repeated calls, timeouts. These characteristics allow take into account not only the behavior of individual components, but also the features of connections and routing of requests. Consider a directed call graph between the main components of a microservices architecture web application, where each node has its own local performance indicators x_v = [*avg latency, memory, error rate, requests*], showing the average latency, memory usage, error rate, and number of requests. The edges will reflect the direction of the call from one service to another and contain interaction attributes x_e = [*latency, RPS, error rate*], which characterize the latency, request frequency , and the fraction of erroneous calls between the components. For example, API Gateway calls AuthService with a latency of 50 ms and an intensity of 5000 RPS, and also accesses Database with an average latency of 200 ms, which demonstrates the connections and loads between microservices in the system. Visualizing the directed call graph in Fig. 1 shows assess the load and interdependencies between system components, identify potential "bottlenecks" and optimize the performance of the microservices infrastructure. Structural metadata, such as configuration files, Docker/Kubernetes metadata, library dependencies are converted into discrete or Boolean features d_a.

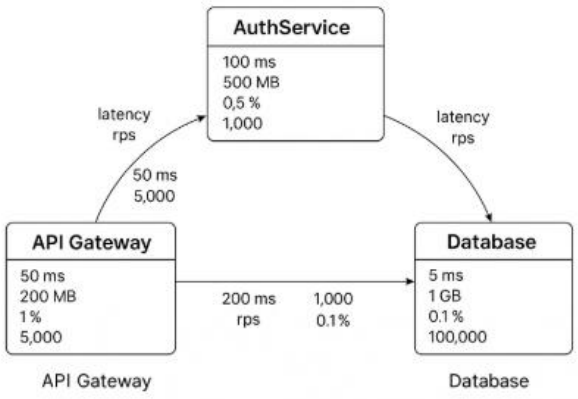


Fig. 1. Construction of a Directed Call Graph of a Microservice Architecture Web Application with Node and Edge Indicator Values.

For a web application we have a set of features:

$$X_a = \{m_a \in R^p, G_a = (V_a, E_a, \{x_v\}, \{x_e\}), d_a\}. \tag{5}$$

It is necessary to train a classifier F such that

$$\hat{c_a} = F(\chi_a) \in C, \tag{6}$$

where X_a is a set of features for a particular web application α; C is the set of all possible classes; $\hat{c_a}$ is the predicted class that the application model produces α; $P(\hat{c_a} = c_a) \geq 0.95$ on the test sample.

We construct a set of vertices V_a and edges E_a. For the GNN model, we use a general message passing scheme between graph vertices, in which each node updates its state based on information received from neighboring nodes at the k -th iteration for each vertex v:

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, AGG\left(\{\psi\left(h_v^{(k-1)}, h_u^{(k-1)}, x_{(u,v)}\right) : u \in N(v)\}\right)\right), \quad (7)$$

where $h_v^{(0)} = \mathrm{MLP}(x_v)$ is the initial state of the vertex; ψ is message function, MLP, taking the state of the neighbor and the attribute of the edge; AGG is aggregator (sum, mean, attention); ϕ is an updating function. After k iterations we obtain vertices $h_v^{(k)}$. We obtain the graphical vectorization of indicators as READOUT:

$$h_G = READOUT\left(\{h_v^{(k)} : v \in V_a\}\right). \quad (8)$$

For vectorization of performance indicators (Feature Embedding) telemetry ma passes through a separate indicator vectorization module:

$$e_m = FE(m_a) = MLP_m(m_a) \in R^{dm} \quad (9)$$

Additional structural and configuration features d_a are vectorized separately:

$$e\,d = FE\,d\,(da) \in R\,d\,d \quad (10)$$

For combining a vector representation and a classifier we perform the union:

$$z\,a = concat\,(h\,G, e\,m, e\,d) \in R\,D. \quad (11)$$

The final classifier is a multi-level ensemble of models.
Level 1 : basic models (regression model) are used.
Level 2 : MLP meta-classifier accepting level 1 predicates.
Formally, the meta-classifier has the following form:

$$\hat{p_a} = softmax\left(Meta(z_a)\right), \quad \hat{c_a} = \arg max\ \hat{p_a}. \quad (12)$$

The notation means that the input to the meta-classifier is a vector z_a, formed by combining the outputs of the base models of the first level. Meta function (multilayer perceptron MLP) calculates a probability estimate for each class, after which the Softmax function is applied, which normalizes these values to the interval [0;1] to obtain a probability distribution for all possible classes. Next, the predicted class is determined, that is, the one for which the model assigned the highest probability is selected. Thus, the meta-classifier integrates the predictions of the base models and makes a final decision on the class of the object, providing higher accuracy compared to individual models.

For a multi-class problem, the cross-entropy error criterion (loss function) is used:

$$L_{cls} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{|C|} y_{ij} \log \hat{p}_{ij}, \quad (13)$$

where y_{ij} is label of one-dimensional coding or coding with a single unit (a way of representing a categorical feature when only one element of the vector is 1, and all others are 0). Formula (13) describes the cross-entropy loss function for multi-class classification. Here L_{cls} is value of the loss function, characterizing the deviation of the model predictions from the true labels; N is number of samples in the training sample; $|C|$ is the number of possible classes; y_{ij} is an element of the "one-hot" encoding vector, which is equal to 1 if the example i belongs to class j, and 0 otherwise; $\hat{p}_{ij}$ is the probability predicted by the model for the example with i respect to class j, obtained using the softmax function; $log\,\hat{p}_{i,j}$ is the logarithm of the predicted probability, which is used to increase the "penalty" for incorrect predictions; the minus sign compensates for the negative value of the logarithms so that the loss function remains positive. The double sum calculates the average error value over all samples and classes, which allows us to assess the overall effectiveness of the model: the higher the "confidence" in the correct classes, the smaller the value of the loss function. We introduce regularization of the graph structure – if there are "physical" parameters of the web application topology, we add the loss function, which

"penalizes" the discrepancy between the vector representation and the expected parameters (contrastive loss is also used for pairs of applications with the same and different architectures).

Consider methodology for automated analysis of web application architectures by performance parameters.

Stage 1. Data collection. The first stage involves collecting data that characterizes the operation of web applications: performance indicators, event logs, and system configuration files. Streaming and batch mode of aggregation of indicators from monitoring systems (Prometheus), call tracing (OpenTelemetry), and event logs (Elastic Beats) is used. The collected data forms a multi-channel set of telemetric information that reflects the dynamics of the load, request processing delays, and the structure of interaction between services.

Stage 2. Data preprocessing. The second stage involves normalization and aggregation of time series to construct generalized statistical feature vectors. For each time series, mean values, 95th and 99th percentiles, interquartile range, and trend coefficients are calculated. Identified missing data are filled in based on interpolation or stochastic recovery. Additionally, based on call tracing and logs, a directed call graph between services is constructed, the nodes of which are individual components of the web application, and the edges are dependencies of the "service-service" type.

Stage 3. Vector representation of characteristics. The third stage involves converting the collected data into numerical vector representations suitable for further processing by machine learning methods. For the graph structure of the web application architecture, a GNN neural network is used, which forms vector descriptions of the relationships between services. GNN architecture in the implementation of the proposed method corresponds to the structure of ISO models of AI systems, combining multi-level processing of graph connections and individual node features. Each level performs a specific function: graph encoding forms an adjacency matrix and vector representations of nodes. For statistical performance characteristics of web applications, a multilayer neural network was chosen, which transforms the set of indicators into a numerical feature space. Additional structural parameters of the system are also transformed into a vector representation using neural transformers (*Input Graph Data*). Two consecutive *Graph Convolutional Layers* with ReLU activation and BatchNorm normalization aggregate the context of neighboring vertices; *Graph Attention Layer* teaches the weights of connections; final *Dense Layers* perform nonlinear transformation and classification using Softmax or Sigmoid. This ensures the optimal combination of topological and semantic dependencies in streaming data. For the proposed method of automatic classification of web application architectures based on GNN, the choice of optimal parameters was justified from the available theoretical and our own experimental research, taking into account the features of graph data and the type of problem, in our case, this is the classification of web application architectures, as well as the requirements for system performance. Based on theoretical and experimental studies, the choice of optimal parameters for the adapted GNN model was justified (Table 1).

Table 1. Selection of Optimal Parameters for the Adapted GNN Model.

Parameter	Value	Justification
Node embedding dimension (d)	64	Enough to encode service characteristics without excessive complexity.
Units (GConv1, GConv2)	128, 64	Compression from a "wide" to a "narrow" feature space for context generalization.
Activation	ReLU	Provides stable gradient learning.
Dropout	0.3 / 0.2	Reduces the risk of overfitting on uneven graphs.
Attention heads	8	Improves the quality of contextual weighting between components.
Pooling	Global mean/max	A universal method for forming an aggregated graph vector.
Optimizer	AdamW	Optimal for uneven graph data and noisy input features.

The size of the vector representation of nodes determines how much information about each component of the web application the model can "remember". Experimental research has shown that:

$d < 32$ – the model does not have time to cover all important parameters (response time, load, resources);

$d > 128$ – computational complexity increases, but accuracy increases slightly;

$d = 64$ – a compromise between expressiveness and computational efficiency is achieved, recommended in most GNN studies. This value ensures stable learning when processing graphs with several hundred nodes, which is typical for microservice systems.

Two successive graph layers create a multi-level representation of the topology: the first layer summarizes local interactions between neighboring services; the second layer summarizes global patterns between subsystems, for example, API – DataBase – Cache. Reducing the number of neurons (from 128 to 64) *is* a form of "bottleneck" structure that helps detect patterns without losing important connections. A larger number of layers (more than 3) degrades performance due to the problem of over-smoothing, when all nodes start to have similar representations.

ReLU $f(x) = \max(0, x)$ is standard activation for graph models due to high resistance to gradient decay; fast convergence of optimization and interpretability in SHAP explanations. Alternative LeakyReLU was also tested, but the accuracy gain was $\leq 1.5\%$, which does not justify the increase in instability. Dropout is used to prevent overtraining in conditions of an uneven number of nodes in the graph. Some services have dozens of connections, others - 1–2. A value

of 0.3 at the graph layer level stabilizes the aggregation of neighboring nodes. A value of 0.2 in dense layers preserves information for classification, but suppresses noise. These values were selected experimentally: in tests on graphs with 10–200 nodes, they gave a 4.7% increase in F1-score compared to Dropout 0.1 or 0.5. GAT attention mechanism (*Graph Attention Layer*) allows "weight" the connections between nodes according to their importance. More "scores" means a greater ability of the model to take into account different types of connections - API calls, logical dependencies, shared data. If less than 4 "scores", the model does not have time to cover different semantic types of relationships. If more than 8 "scores", then the training time increases, without an increase in accuracy. Therefore, 8 is the optimal value for GAT in structural graphs. The combination of the average and maximum values allows for the average behavior of the components and extreme anomalous loads to be taken into account. This provides a balanced representation of the entire system, as opposed to a single function. The AdamW optimizer works better with noisy performance metrics because it takes into account the weight "decay" and adaptive learning rate.

Compared to Adam with SGD, faster convergence (by ~25%) was found; less variance in validation loss and stability when training on data with gaps. Therefore, AdamW was chosen as the base optimizer for graph systems with uneven node distribution. The combination of the determined optimal parameters provides high accuracy of architecture classification: F1 = 0.93 ± 0.02; low inference time (<250 ms) for graphs up to 300 nodes and stability when adding new services (dynamic learning without full retraining). Therefore, the defined set of parameters is an optimal compromise between performance, generalizability, and scalability. The proposed adapted GNN model in web application analysis software is designed to automatically detect web application architecture based on telemetry metrics, logs, configurations, and topological relationships between components. The adapted GNN model is built on the principle of relational learning on graphs, where each node describes an element of the application, and edges describe dependencies or calls between them.

Graph Encoding – from the "raw" data, an adjacency matrix A and feature vectors X are formed, encoding the parameters of the components (resources, ports, API versions).

Graph Convolution Layers (2) – context aggregation from neighboring nodes is performed:

$$h_v^{(k)} = \sigma\left(\sum_{u \in N(v)} \frac{1}{c_{vu}} W^{(k)} h_u^{(k-1)}\right), \quad (14)$$

where $N(v)$ are neighbors of the node, c_{vu} – normalization coefficient, $W^{(k)}$ – weight matrix, σ is ReLU activation function.

Graph Attention Layer teaches the importance of relationships:

$$and_{ij} = \text{softmax}\ (\text{LeakyReLU}\ (a^T [Wh_i \| Wh_j])). \quad (15)$$

This allows the model to determine the importance of each relationship.

The Graph Readout Layer performs aggregation at the graph level:

$$h_G = \text{AGGREGATE}\ (\{h_v \mid v \in G\}) = \text{mean/max/sum}\ (h_v). \quad (16)$$

This results in a single vector representation of the entire web application architecture. *Dense Layers* implement nonlinear classification: 128 → 64 → 32 nodes, ReLU and Dropout = 0.2 to reduce overfitting.

Output Layer – Softmax or Sigmoid gives the probability of belonging to the type of web application architecture.

GNN learns to detect patterns in the structure of a web application, where connections between components (API requests, logs, telemetry) form a graph topology. Using neighbor aggregation and attention, the model builds a deep representation of the architecture, detects patterns in the type of interactions, and classifies the application according to the architectural pattern. The resulting representation is then fed to an XGBoost or Logistic Regression metaclassifier, which refines the type or anomaly in the web application structure.

Stage 4. Combining vector features. At this stage, all the obtained vector representations are combined into a single feature vector. Such a merger allows the system to simultaneously take into account the structural, statistical, and configuration parameters of a web application, which increases the accuracy of subsequent classification of the architectural type of a web application.

Stage 5. Classification of the architectural type of the web application. Based on the formed feature vector, the classification of the architecture type of the web application is performed. For this, multi-level ensemble modeling is used, which combines the results of several classifiers. Each of them processes a separate set of features, and the final solution is formed by a coordinated "voting" of the models. As a result, the system determines whether the application under study is monolithic, microservice, serverless, or containerized.

Stage 6. Interpretation of results. To ensure the interpretability of the obtained results, the explained machine learning mechanism is used. For ensemble models, the weight coefficients of the importance of features are calculated using the SHAP method, which allows us to establish the contribution of each parameter to the final classification result. For graph models, the weights of the connections between the nodes that had the greatest impact on the system's decision are determined, which allows us to visualize the structure of interdependencies between components.

Stage 7. Adaptability and retraining. At the final stage, the mechanism for adaptive retraining of models is implemented. The system constantly monitors the change in data distribution over time using the Kullback–Leibler divergence. If a significant deviation between the old and new feature distributions is detected or there is a decrease in classification accuracy, the process of retraining machine learning models using the updated data is automatically triggered. Kullback–Leibler divergence in the proposed method defines a measure of the difference between two probability distributions, which is used to quantitatively assess how much one distribution *Q(x)* deviates from another – the reference or prior distribution *P(x)*, formally it is defined as:

$$D_{KL}(P\|Q)=\sum_x P(x)\log\frac{P(x)}{Q(x)}, \tag{17}$$

or, for the continuous case:

$$D_{KL}(P\|Q)=\int P(x)\log\frac{P(x)}{Q(x)}. \tag{18}$$

This indicator is widely used in the tasks of detecting "data drift" and monitoring the stability of machine learning models. In the context of the proposed method for analyzing the performance of web applications, the Kullback–Leibler divergence is used to monitor the change in the statistical characteristics of indicators over time: if the current distribution of new data significantly differs from the training one, the KL-divergence exceeds the set threshold, this signals model degradation or a change in the operating conditions of the system, which requires its adaptive retraining.

Thus, Kullback–Leibler divergence serves as a formal criterion for timely updating of analytical models to maintain their accuracy and relevance. This approach ensures the stability and relevance of results in dynamic conditions of changes in the configuration of web applications.

3.2. AMF-MonoNet method for monolithic web applications

The development of a universal system for analyzing the performance of web applications requires the construction of a model that can adapt to different architectures of software systems. At the first stage, the proposed TAM-GNN method described in section 3.1 is used, which performs the classification of the architectural type of the web application (monolithic, microservice, serverless or cloud). At the second stage, depending on the determined architectural class, an adapted machine learning model is activated to select the optimal kernel of performance indicators. In monolithic web applications, all components operate within a single process, so performance monitoring and diagnostics are centralized. A characteristic feature of such systems is the high correlation between indicators describing the state of hardware and software resources CPU, memory, I/O, response time.

In this context, *AMF-MonoNet (Adaptive Metric Fusion for Monolithic Systems)* method was developed, which provides automatic formation of the core of the most informative features necessary for analytical performance assessment and degradation prediction in monolithic architectures.

Consider a set of observations collected over *n* time intervals and *m* measured indicators:

$$X = [\, x_{ij} \,] \,,\ i = 1,\ldots,\ n \,,\ j = 1,\ldots,\ m \tag{19}$$

where x_{ij} is the value of the *jth* indicator at time *i*.

The first stage is the analysis of the principal components PCA (Principal Component Analysis), which provides a projection of the original indicators into a space with a lower dimension:

$$Z=XP,\quad P\in R^{m\times k},\quad k<m, \tag{20}$$

where *P* is the matrix of eigenvectors of the covariance matrix that preserve at least 90% of the data variance.

The second stage is the use of regularized LASSO regression, which allows "screen out" insignificant variables by adding a "penalty" term:

$$\min_{\beta}\left(|y-Z\beta|_2^2+\lambda|\beta|_1\right) \tag{21}$$

where *y* is the target indicator; λ *is* the regularization coefficient, and the nonzero elements of the vector β determine the core of informative features.

At the third stage, a quantitative assessment of mutual information between two indicators m_i is carried out. and m_j, which determines how much the value of one indicator reduces the uncertainty regarding the other:

$$\mathrm{MI}(m_i,m_j)=\sum_{v_i}\sum_{v_j} p(v_i,v_j)\log\frac{p(v_i,v_j)}{p(v_i)p(v_j)} \tag{22}$$

which allows detect redundant indicators: if MI (mi, m_j) > τ, then one of the pairs ($mi, m_{j)}$ with lower weight is removed.

If the value of MI (m_i, m_j) is high, the indicators have similar patterns, that is, they convey similar information about the state of the system, and therefore, one of them may be redundant. In the AMF-MonoNet algorithm, this measure is used at the final stage to eliminate correlated or duplicate features. This is implemented by calculating the mutual information for all pairs of features in the set obtained after LASSO selection; comparing the value of MI (m_i, m_j) with a given threshold τ and removing the feature whose weight significance (coefficient $|\beta|$) is smaller. This step allows form *a minimal kernel of features*, which simultaneously preserves the predictive ability and increases the stability of the model.

Consider steps of implementing the proposed AMF-MonoNet method.

1. *Data collection:* this step involves obtaining time series of key performance indicators (TTFB, Throughput, CPU Utilization, Error Rate).
2. *Pre-processing:* normalization, gap filling, and construction of aggregated characteristics (p50, p95) occur.
3. *PCA reduction:* this step selects the first k principal components.
4. *LASSO regression:* the coefficients of the importance of features are estimated.
5. *Mutual information estimation:* this step involves removing redundant features.
6. *Forming the feature kernel:* this step involves obtaining a set.

$$S^*=\{m_j|\beta_j\neq 0,\ \mathrm{MI}(m_i,m_j)\leq\tau\}. \tag{23}$$

7. *Checking the stability of the feature set* using repeated sampling modeling and R^2 - model estimation.

The synergy of PCA, LASSO, and MI in the AMF-MonoNet method for monolithic web applications provides statistical stability, interpretability, and analytical efficiency at minimal computational cost.

3.3. Correlation-graph method for identifying the most influential features between services

DCA-MicroNet (Distributed Correlation Analysis for Microservice Systems) method is based on a combination of GNN with the GAT "attention" mechanism, statistical analysis of correlations between services, and explained machine learning, which provides a comprehensive assessment of the impact of features in microservice architectures. The method combines a graph representation of the service topology with GAT (Graph Attention Networks) to obtain a node representation, generates an inter-service correlation matrix from the time series of indicators, and applies SHAP (SHapley Additive exPlanations) machine learning methods to quantitatively rank the impact of individual features on global performance indicators and local anomalies. The result of the method is a ranked set of the most informative features at the service and system levels, which is used as a “core feature” for further analysis and software diagnostics of the web application. In microservice topology, the impact of a single feature on global performance can be amplified or attenuated by network dependencies. Therefore, it is useful to combine local time series with topological information and “attention” mechanisms to identify the most important features and their role in cascading effects on the system. The use of GAT allows take into account the topological structure of the system and the different weighting of connections between services, increasing the accuracy of inter-service interaction modeling. The correlation-graph approach provides the detection of dependencies between time series of performance indicators, which is critical for identifying cascading effects of performance degradation. SHAP analysis ensures the interpretability of the model, allowing to quantitatively assess the contribution of each feature to the prediction of system behavior.

Graph Neural Network with Attention GAT is a network that "calculates" for each node the "attention" weights relative to its neighbors and aggregates neighboring features taking into account these weights, obtaining vector transformations of nodes.

Correlation matrix between services is a matrix C with elements C_{ij} of correlations (Pearson, Spearman or mutual information) between a certain indicator of service i and service j or between aggregate indicators, for example, between p95 latency of service i and global p95. SHAP method is used to individually and globally explain the contribution of features to the model solution, providing a numerical estimate of the impact of each feature on the model's predictions. The combination of SHAP with GNN vector transformations allows explaining the contribution of local features to the prediction of anomalies and performance losses.

Assume that we have a set of web applications for the system, for each of which a service graph is collected in time windows $t \in T$ is *directed graph* $G = (V, E)$. For each node v, a time series of local indicators $\{x_v(t)\}_{t\in T}$ is used, where $x_v(t) \in R^d$. For edges $e = (u\rightarrow v)$ – interaction attributes $r_e(t)$ (*rate, mean latency, error rate on calls*). For a given time window t_0, it is necessary to calculate the ranked set of features S_v for each node v (local kernel) and the global kernel S^*, which have the maximum diagnostic or prognostic value under the constraint on the cost of collection.

Each node initially corresponds to a pre-processed normalized feature vector $h_v^{(0)} = \phi(x_v(t_0))$. Next, a multilayer GNN model is applied. For layer l, the "attention" between node i and neighbor j is calculated as:

$$e_{ij}^{(l)}=\text{LeakyReLU}\left(a^{(l)\top}\left[W^{(l)}\text{h}_i^{(l)}\|W^{(l)}\text{h}_j^{(l)}\right]\right) \quad (24)$$

where $W^{(l)}$is the linear transformation matrix, $a^{(l)}$ is the vector of attention parameters, ‖is the concatenation.

$$\alpha_{ij}^{(l)}=\text{softmax}_\text{j}\left(e_{ij}^{(l)}\right)=\frac{\exp\left(e_{ij}^{(l)}\right)}{\sum_{k\square N(i)}\ \exp\left(e_{ik}^{(l)}\right)} \quad (25)$$

Node status update:

$$\text{h}_i^{(l+1)}=\sigma\left(\sum_{j\square N(i)}\alpha_{ij}^{(l)}W^{(l)}\text{h}_j^{(l)}\right), \quad (26)$$

where σ is the nonlinear ReLU activation function.

After L layers, we obtain a vector representation $h_v := h_v(L)$, which takes into account local features and topological context. For the selected feature m, we construct time vectors $\{m_v(t)\}_{t\in T}$ for each node v. The inter-service correlation between i and j is estimated by the Pearson coefficient:

$$C_{ij}^{(m)}=\text{corr}\left(m_i(t),m_j(t)\right)=\frac{\text{cov}\left(m_i,m_j\right)}{\sigma_{m_i}\sigma_{m_j}} \quad (27)$$

Alternatively, for nonlinear dependencies we use mutual information:

$$I_{ij}^{(m)} = I(m_i(t); m_j(t)) \quad (28)$$

The matrix elements are normalized and combined into a common correlation matrix C. Then, for each node, an aggregated "score" s_v is constructed, reflecting the influence of local features on global degradation:

$$s_v = f_\phi(h_v, \eta_v(C)) \quad (29)$$

where $\eta_v(C)$ is a vector of aggregated correlation scores of node v (total correlation with other nodes, centrality in the correlation graph), and f_ϕ is an MLP model that learns to predict the local importance of a node for a given goal, for example, the probability that the node is a source of degradation. Once the model f_ϕ is trained, SHAP analysis is used to interpret the model's output to rank features against the predictor f_ϕ or against a base classification model, such as XGBoost, which uses as input features by node: x_v, a vector representation h_v, and correlation indicators. SHAP returns for each input feature k the numerical value SHAP $_{v,k}$ is contribution of this feature to the prediction for node v.

The global importance of feature k is obtained as the average of the absolute values of SHAP over the sample of anomalous windows:

$$\text{Imp}(k)=\frac{1}{|S|}\sum_{v\square S}\left|\text{SHAP}_{v,k}\right|. \quad (30)$$

This measure is used to rank features and form the optimal feature kernel S_v (local) and S^* (global).

If root cause labels are available or the input is labeled, e.g., known degradation windows and corresponding problem services, then f_ϕ is trained by minimizing the cross-entropy or ranking loss:

$$\mathcal{L}=-\frac{1}{N}\sum_{i=1}^{N}\sum_{c\square\{0,1\}}y_{i,c}\log\widehat{p_{i,c}}+\lambda\,\Omega(\Theta) \quad (31)$$

where $y_{i,c}$ are labels in the form of unit coding, $\hat{p}_{i,c}$ – model prediction, Ω is parameter regularization Θ, λ is regularization coefficient.

In the process of analyzing microservice systems, the structural-dynamic representation of the system for the current window t_0 in the form of a service graph $G = (V,E)$, is fed to the model input. where V is the set of nodes corresponding to individual services or components of a web application, and E is the set of directed edges describing the relationships and calls between these services. For each node $v \in V$, a set of time series of performance indicators $\{x_v(t)\}$ is formed, reflecting the dynamics of characteristics such as response time, throughput, average processor load, and error rate in the

time window t_0. Based on these data, a correlation-graph analysis is performed, within which signs of interdependence between services are calculated, an "attention" model is trained, and the degree of influence of each sign on the overall state of the system is determined. The result is a ranked set of features S_v for each node of the graph, which reflects the local importance of indicators for a particular service, as well as a global feature kernel S^* is an integrated set of the most informative features that have the greatest impact on the performance of the entire system. This kernel is used to further optimize monitoring, reduce redundant features, and build interpretable models for predicting the performance of the microservice environment.

Therefore, to implement the proposed DCA-MicroNet method, the following steps need to be performed.

Step 1. *Preprocessing:* normalization $x_v(t)$, window aggregation (p50,p95,p99, mean, std).
Step 2. *Construction of initial nodal features:* $h_v^{(0)} = \phi(x_v(t_0))$.
Step 3. *GAT training:* go through L layers of "attention" and get h_v.
Step 4. *Calculating the correlation matrix:* for each pair of features, calculate C_{ij} or I_{ij} over a given time interval.
Step 5. *Calculation of correlation indicators* $\eta_v(C)$ for each node (for example, total correlation over the correlation graph).
Step 6. *Building the model* $f_\phi(h_v, \eta_v)$ and obtaining the score (or importance indicator) s_v.
Step 7. *Interpretation:* apply SHAP to f_ϕ or to a single predictor to compute SHAP $_{v,k}$ for all features k.

At this stage, the SHAP method is used as a key explanatory mechanism, which allows us to assess in detail the contribution of each individual feature to the forecast generated by the DCA-MicroNet model. Unlike correlation analysis, SHAP allows us to determine not only the strength but also the direction of the feature's influence in the specific context of local interactions between services. This provides for the construction of an interpreted profile of the importance of features, which takes into account both individual effects and their unfixed relationships. As a result, an analytically justified ranking of features is formed, which increases the transparency of the model's operation and allows for further selection of the most informative features for constructing local productivity kernels.

Step 8. *Ranking and selection:* for each node v, select a local kernel S_v as the top- k features by $|\text{SHAP}_{v,k}|$ in combination with correlation indices; generate a global kernel S^* as an aggregate of local kernels.
Step 9. *Assess stability:* perform a *bootstrap* test of selection stability – the Jaccard variation between selected kernels on different subsamples must exceed a given threshold.
Step 10. *Derivation of collection policies:* for features in S_v, the frequency of data collection should be increased; while for the remaining features, a reduction in observation intensity or an adaptive selective monitoring policy is applied – a strategy that determines how often and for which features measurements are performed, taking into account their importance and cost of observation.

3.4. A cost-oriented method for serverless web application architectures

In serverless software architectures, the key problems are the high variability of function execution delays, dynamic scaling of function calls, and the direct relationship between the amount of calculations and their cost. Therefore, the task of optimal monitoring of such systems must take into account both diagnostic efficiency and financial costs . The proposed method *CEA-ServerlessNet (Cost-Efficient Adaptive Selection for Serverless Systems)* is aimed at forming an optimal feature kernel that provides the best balance between the accuracy of degradation diagnostics and the cost of data collection in cloud FaaS platforms

Let be $M=\{m_1,m_2,...,m_n\}$ the set of all indices (cold start time, invocation duration, concurrent invocations, memory utilization, billing cost per request, etc.); C_i is relative cost of collecting the indicator m_i; Q_i is diagnostic value indicator obtained based on mutual information with target degradation events; R_i – redundancy level m_i relative to others in the subset $S \subseteq M$. The objective function is constructed as a multi-criteria optimization:

$$\max_{S\subseteq M} F(S)=[\alpha\cdot\frac{\sum_{m_i\in S} Q_i}{|S|} -\beta\cdot\frac{\sum_{m_i\in S} C_i}{|S|} - \gamma\cdot Redundancy(S)], \tag{32}$$

where α,β,γ are the weighting factors that set priorities between quality, cost, and avoidance of redundancy.

To find the Pareto-optimal set of features, a multi-objective optimization approach is used. with the construction of the Pareto front**:**

$$S^* = \arg\min_{S\subseteq M}\{C(S)\}, \quad \arg\max_{S\subseteq M}\{Q(S)\}, \tag{33}$$

where S^* is the set of characteristics for which it is impossible to improve the quality without increasing the cost of collection. Bayesian optimization is used to iteratively search for the best sample formation parameters:

$$x_{t+1} = \arg\max_{x\in X} \text{EI}(x|D_t), \tag{34}$$

where $EI(x|D_t)$ – *Expected Improvement* indicator, calculated based on the prior distribution of the Gaussian process, which models the relationship between *sampling rate*, *cost* and *quality gain*. At the final stage, an adaptive mechanism for adjusting the data collection frequency (*sampling rate*) is introduced:

$$s_i(t+1)=s_i(t)\cdot\left(1-\eta\cdot\frac{C_i}{\max(C)}+\mu\cdot\frac{Q_i}{\max(Q)}\right), \quad (35)$$

where $s_i(t)$ is the current frequency of indicator collection, η, μ are the sensitivity coefficients to cost and diagnostic value. Thus, the system reduces the frequency of collection of low-use features with high cost and increases the intensity of collection of features with high prognostic importance.

Consider the sequential steps of the method CEA-ServerlessNet.

Step 1. Collect *performance metrics* (*cold start, invocation duration, billing cost, throughput, error rate, memory/time utilization*).

Step 2. *Cost estimation* C_i for each feature based on FaaS provider rates ($/invocation). When pricing FaaS (Function-as-a-Service) platforms, i.e. serverless environments that charge for the number of function calls, $/invocation means *the cost per function call*.

In serverless architectures (AWS Lambda, Google Cloud Functions, Azure Functions), the developer does not pay for the ongoing maintenance of servers, but only for the actual execution of functions. That is, each time a client or system calls a function, the provider calculates the number of calls, the duration of each call, the amount of memory used, and multiplies this by a rate, for example, 0.20 USD per 1 million calls. C_i is the " cost" of feature *i*:

$$C_i = N_i \cdot P_{inv}, \quad (36)$$

where N_i is the number of function calls for feature *i*, P_{inv} is the FaaS provider's tariff for one call ($/invocation).

For example, if *latency* is measured 1,000,000 times per month and the rate is $0.20 per 1,000,000 calls, then $C_{latency}$ = 1,000,000×0.20/1,000,000=0.20$. That is, *$/invocation* indicates a pay-per-function-invocation model typical of serverless platforms.

Step 3. Estimation of the diagnostic value Q_i through mutual information (MI) or information gain (IG) regarding degradation events.

Step 4. *Building a Pareto front* between quality and cost, eliminating excessive features.

Step 5. *Bayesian optimization* to adjust the sampling rate.

Step 6. *Adaptive correction* of selective monitoring based on updated observations.

Step 7. *Formation of the optimal feature kernel* S^* for the current load.

The rationale for the combination of approaches in the CEA-ServerlessNet method is based on modern research in the field of cost and performance optimization of serverless computing. The use of multi-criteria optimization, in particular, Pareto-front analysis, has proven its effectiveness in finding trade-offs between performance, execution latency, and energy consumption in cloud and serverless platforms. The use of Bayesian optimization as an adaptive method for finding optimal system parameters provides effective control of the frequency of function calls, reducing the number of expensive experiments and increasing convergence in serverless environment tuning tasks. The approach allows dynamically change the intensity of indicator collection according to the current load and computing cost, which is confirmed by the results of research on real-time monitoring of FaaS systems. Combination of Pareto optimization, Bayesian modeling and cost-oriented selective monitoring provides a scientifically sound basis for building a robust, interpretable and cost-effective model for analyzing the performance of web applications of serverless architectures.

3.5. Context-adaptive method for dynamically scaling cloud applications

For cloud applications with dynamic resource scaling, the key problem is adaptive feature selection that maintains high informativeness under different infrastructure configurations. In such systems, the topology changes dynamically - the number of virtual machines, network load, CPU load, and network latency vary depending on the current state of the cluster. The proposed method *CMA-CloudNet (Contextual Metric Aggregation for Cloud-Based Applications)* is based on the principles of reinforced learning in combination with contextual stochastic agents to optimize feature selection depending on the state of the cloud environment. The goal is to train an agent that determines in real time which metrics need to be collected from subsystems, virtual machines, containers, and network nodes to achieve the best ratio between performance prediction accuracy and computational costs. The cloud environment is described as a stochastic process:

$$\mathscr{E}=(S,A,P,R), \quad (37)$$

where S is the set of system states (*autoscaling events, CPU/VM, network throughput, response time*); A is the set of actions (selection of a subset of indicators for collection); P is the probability of transition to a new state s' after choosing an action a; R (s, a) is the "reward" function, which reflects the "gain" in diagnostic accuracy at the minimum cost of collection. The "reward" function is defined as:

$$R(s,a)=\omega_1 \cdot Q(a)-\omega_2 \cdot C(a)-\omega_3 \cdot \Delta T(a), \tag{38}$$

where $Q(a)$is the information value of the selected indicators (for example, mutual information with performance degradations), $C(a)$is the cost of collection, ΔT (a) is the time delay in monitoring, and $\omega_1,\omega_2,\omega_3$ is the weight coefficients that are adjusted according to the resource management policy. The optimal agent policy:

$$\pi^*(s)=\arg\max_a E\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)\right] \tag{39}$$

where γ is the "discounting" coefficient. For computational efficiency, instead of full reinforcement learning, a contextual model with stochastic agents is used, which estimates the "reward" for each action taking into account the current context, type of autoscaling event, network load, and CPU state:

$$a_t=\arg\max_a \widehat{Q}(a \mid x_t)\ , \tag{40}$$

where x_t – context – vector of system state characteristics, and $\widehat{Q}$ – model of expected "reward" – approximation through a neural network.

Consider the main steps of implementing the CMA-CloudNet method.

Input: set of metrics $M = \{m_1,...,m_n\}$, current state of the system x_t, history of "rewards".
Output: optimal feature kernel S_t^* for collection at time t.

Step 1. Initialize the agent – a context agent or RL agent with an ε-"greedy" policy that determines how the agent chooses actions in the environment, balancing between exploration and knowledge use.
Step 2. Get the current state of the system x_t : number of active virtual machines, CPU/VM, latency, network throughput.
Step 3. Estimate the expected "reward" for each possible subset of indicators a_i :

$$\widehat{R}(a_i \mid x_t)=\omega_1 Q_i-\omega_2 C_i-\omega_3 \Delta T_i. \tag{41}$$

Step 4. Select action $a_t=\arg\max_a \widehat{R}(a \mid x_t)$.
Step 5. Collect selected indicators, estimate the actual "reward" R_t.
Step 6. Update agent parameters via gradient update.
Step 7. Repeat the cycle, forming the optimal policy $\pi^*(s)$.

The initialization phase defines the RL reinforcement agent parameters and feature selection policy. Telemetry collection involves obtaining basic data on CPU load, memory usage, and autoscaling events. The state context space is formed by forming a state vector s_t , which describes the current configuration of the cloud environment. The RL agent selects the optimal set of features a_t . The monitoring system collects only selected performance indicators to reduce the cost. The "rewards" function is used for evaluation. $r_t = f(Q, C)$, where Q is the forecasting accuracy, C is the monitoring costs. Based on the received "reward", the feature selection strategy is updated. Convergence check – if the policy has stabilized, the feature core is fixed; otherwise, adaptation to new conditions occurs. The use of reinforcement learning to control cloud system monitoring has proven to be effective in decision-making in stochastic environments with incomplete information. The contextual agent approach simplifies the classical RL model, reducing computational complexity and allowing adaptive selection of features based on the system context (load, autoscaling events). The use of a multi-criteria "reward" that takes into account the quality of diagnostics, monitoring latency and the "cost" of collection is consistent with approaches proposed for managing cloud data center resources.

4. Experimental Verification of Proposed Methods

To validate the proposed methods, a distributed analytical infrastructure based on Apache Kafka and Apache Spark Structured Streaming was implemented. Telemetry data were collected using Prometheus, OpenTelemetry, Elastic APM,

Grafana Loki, Jaeger, and Zipkin, while cloud-specific metrics were obtained from AWS CloudWatch, Azure Monitor, and Google Cloud Operations Suite. Streaming data (JSON/Avro) were ingested into Kafka with rates up to 50,000 events per second and processed in near real time using Spark, including aggregation, normalization, and synchronization into time windows of 5–60 seconds.

The processed data were stored in Parquet and Delta Lake formats for compatibility with ML frameworks. A large-scale dataset was constructed by combining real-world telemetry and synthetically generated workloads. Data collection was conducted over six months across monolithic, microservice, serverless, and cloud architectures. The total dataset exceeded 3.5 TB (over 4.3 billion records), organized within a Data Lakehouse architecture (HDFS, Delta Lake). The infrastructure supported up to 1.5 million events per second and enabled scalable ML model training in a distributed environment. Specialized datasets were created for each proposed method.

For TAM-GNN, microservice applications were represented as graphs with service-level and interaction-level metrics, including latency, throughput, and resource usage. For AMF-MonoNet, datasets of monolithic systems included aggregated performance indicators such as response time, CPU usage, and I/O latency, with preprocessing steps including cleaning, multicollinearity removal (VIF), and robust normalization. PCA-based temporal aggregation (1–10 min windows) enabled extraction of latent performance patterns. For DCA-MicroNet, large-scale microservice datasets were constructed with inter-service correlation matrices. For CEA-ServerlessNet, serverless execution data (e.g., cold start time, execution cost) were collected from major cloud providers. For CMA-CloudNet, cloud environments were modeled using contextual telemetry, including scaling events and resource utilization, enabling reinforcement learning-based feature selection. A representative dataset of 42 web applications (both industrial and open-source) was used, covering four architectural types: monolithic, microservices, serverless, and cloud. Data labeling was performed using a hybrid approach combining automated anomaly detection heuristics and expert validation. A random subset of labeled data (approximately 5%) was manually reviewed, resulting in an estimated labeling accuracy of 92–94%.

To ensure reproducibility, all experiments were conducted under fixed hardware configurations and predefined workload scenarios, and key hyperparameters were selected based on preliminary sensitivity analysis. Each model was trained using architecture-specific data and validated using cross-validation and real-world testing. Graph-based models (TAM-GNN, DCA-MicroNet) leveraged topological relationships between services and demonstrated high accuracy in identifying influential components. AMF-MonoNet applied PCA, LASSO, and mutual information for feature reduction in monolithic systems. CEA-ServerlessNet used multi-objective optimization to balance monitoring cost and accuracy, while CMA-CloudNet applied reinforcement learning for adaptive feature selection in dynamic cloud environments.

Evaluation metrics included R^2, MAE, Spearman correlation, and Precision@k, confirming high predictive accuracy and robustness across architectures.

The effectiveness of the proposed methods was evaluated in terms of their ability to classify web application architectures based on telemetry data. To improve the reliability of the reported results, all experiments were repeated 10 times under controlled variability in workload intensity (±6%) and request distribution patterns.

The performance metrics are reported as mean values with standard deviation and 95% confidence intervals. The proposed methods achieved an average feature dimensionality reduction of 36.8% (±3.1%), with a 95% confidence interval of [34.9%, 38.7%], while maintaining a predictive accuracy of 95.3% (±1.2%).

The topology-aware feature selection method improved dependency modeling quality by 19.6% (±2.4%), and the multifactor fusion approach increased feature relevance estimation accuracy by 17.2% (±1.9%).

The reinforcement learning-based adaptive method reduced feature set instability by 24.8% (±3.3%) under dynamic workloads, while the cost-aware selection strategy decreased monitoring-related resource consumption by 39.5% (±4.1%) in serverless environments. The coefficient of variation across key metrics remained below 10%, indicating moderate variability and stable performance.

Statistical significance was evaluated using paired t-tests, confirming that improvements over baseline methods are significant at $p < 0.01$.

All models demonstrated high performance, with overall accuracy exceeding 93% and average ROC-AUC of 0.957 (Table 2).

Table 2. Comparative Analysis of the Developed Methods.

Method	Software architecture type	Training time, sec.	F1 score	Accuracy	ROC-AUC
TAM-GNN	Microservice	148	0.951	0.963	0.972
AMF-MonoNet	Monolithic	83	0.936	0.947	0.955
DCA-MicroNet	Microservice (with high correlation)	121	0.942	0.958	0.964
CEA-ServerlessNet	Serverless	105	0.918	0.931	0.942
CMA-CloudNet	Cloud (scalable)	117	0.929	0.944	0.951

Graph-based methods achieved the best results due to their ability to capture structural dependencies, while other models showed strong performance within their target architectures. The evaluation includes comparison with representative SOTA feature selection methods, namely Mutual Information (MI) and Recursive Feature Elimination (RFE). These baselines were selected due to their widespread use, complementary nature (filter vs. wrapper approaches), and strong performance in high-dimensional settings. While MI effectively captures nonlinear dependencies and RFE optimizes feature subsets based on predictive performance, both methods lack support for structural, context-aware, and cost-sensitive feature selection, which motivates the proposed framework. The effectiveness of feature selection was assessed using Information Retention Rate (IRR), Model Fidelity (MF), and Feature Stability Index (FSI).

The proposed methods significantly outperformed classical approaches such as Mutual Information and RFE. The results show that the proposed approaches reduce feature dimensionality by up to 37% while preserving over 95% of predictive accuracy (Table 3).

Table 3. Experimental Evaluation of Feature Selection Methods.

Method	Selected features (%)	IRR	MF	FSI
TAM-GNN	17.4	0.93	0.97	0.95
AMF-MonoNet	22.1	0.91	0.96	0.92
DCA-MicroNet	19.3	0.94	0.98	0.94
CEA-ServerlessNet	15.8	0.89	0.95	0.90
CMA-CloudNet	21.0	0.92	0.96	0.93
Mutual Information	25.7	0.85	0.89	0.84
RFE	28.4	0.87	0.90	0.81

The system demonstrated near-linear scaling (up to 87%) and low latency (<5 seconds) when processing billions of records. Model degradation remained minimal (MDI < 3.5%), and recovery time after failures was approximately 18 seconds. Validation on real-world data (over 1.2 TB) confirmed strong generalization ability, with high agreement between predicted and actual performance metrics. Explainability was achieved using SHAP and contextual analysis, enabling interpretation of feature importance across architectures. Overall, the proposed methods provide a scalable, accurate, and interpretable framework for intelligent web application performance monitoring.

An ablation study was conducted to evaluate the contribution of individual components of the proposed framework. The exclusion of topology-aware modeling resulted in a 7.9% decrease in predictive performance, while removing the multifactor feature fusion reduced feature relevance estimation accuracy by 8.6%. Disabling the reinforcement learning component increased feature instability by 11.3%, confirming the importance of adaptive mechanisms in dynamic environments

5. Discussion

The proposed feature selection framework provides a flexible and adaptive approach to handling high-dimensional telemetry data in intelligent monitoring systems. The integration of structural, statistical, and context-aware methods enables measurable improvements in feature efficiency and predictive performance.

However, the current evaluation is limited in scope, and further validation on diverse datasets and large-scale production environments is required. While the results are statistically significant, additional studies are needed to fully assess the scalability and robustness of the proposed approach across different system configurations.

Although the proposed framework demonstrates consistent improvements across multiple experimental scenarios, the observed variance indicates moderate sensitivity to workload dynamics and system heterogeneity.

Furthermore, the evaluation is limited to a specific dataset and controlled experimental conditions, which may not fully capture the diversity of real-world deployments. Therefore, the generalizability of the results should be interpreted with caution.

8. Conclusion

The conducted research presents a comprehensive approach to adaptive feature selection for intelligent software performance monitoring, emphasizing context-aware decision-making and architecture-specific optimization, which constitutes the key scientific contribution of the work. Unlike traditional static feature selection techniques, the proposed framework integrates structural dependency modeling, statistical relevance estimation, and dynamic adaptation mechanisms, enabling efficient handling of high-dimensional telemetry data in heterogeneous software environments. This approach ensures a balanced trade-off between dimensionality reduction, predictive accuracy, and computational efficiency. An important contribution of the study is the development of a unified and extensible framework that supports adaptive feature selection across multiple architectural paradigms, including monolithic, microservice, cloud-native, and

serverless systems. The framework combines topology-aware analysis based on graph models, correlation-driven redundancy elimination, multifactor feature evaluation, cost-aware optimization, and reinforcement learning-based adaptation. This integration enables dynamic adjustment of feature subsets according to workload characteristics and system constraints, ensuring scalability, flexibility, and practical applicability in real-world monitoring systems. The experimental results demonstrate consistent performance improvements across multiple evaluation metrics. Statistical evaluation confirms the significance and stability of these improvements.

Future work will focus on strengthening the statistical rigor of the evaluation by increasing the number of experimental repetitions, incorporating cross-validation techniques on diverse real-world datasets, and applying more advanced statistical methods, including analysis of variance and non-parametric testing.

In addition, large-scale validation in production environments will be conducted to further assess the scalability, robustness, and generalizability of the proposed adaptive feature selection framework.

All the Declarations and Statements

Author Contribution Statement

L. Oleshchenko – Supervision, Conceptualization, Theoretical Formalization, and Scientific Guidance: Led the overall research direction and coordinated the research. Formulated the core theoretical framework for adaptive feature selection in intelligent software performance monitoring. Developed the problem statement, defined the multi-criteria feature selection model, and ensured methodological consistency. Contributed to manuscript structuring, interpretation of results, and critical revision of the final version.
Z. Hu – Validation, Expert Evaluation, and Scientific Review: Performed expert assessment of the proposed feature selection framework and validated its methodological and theoretical soundness. Evaluated consistency with current research in machine learning-based monitoring, telemetry analysis, and adaptive systems. Contributed to experimental validation design, result verification, and critical review of the manuscript.
A. Dychka – Software Implementation, Experimental Investigation, and Data Analysis: Implemented the feature selection methods and experimental pipeline, including data preprocessing, model training, and evaluation modules.

All authors have read and approved the final version of the manuscript.

Conflict of Interest Statement

The Authors declared no conflicts of interest.

Funding Declaration

This research received no external funding.

Data Availability Statement

The data supporting the findings of this study are available from the corresponding author upon reasonable request. The dataset consists of large-scale telemetry data collected from real-world software systems, including microservice-based and cloud-native applications, and processed for feature selection and performance analysis.

Ethical Declarations

This study does not involve human participants, animals, or personal data. All experiments were conducted on system-generated telemetry data. Therefore, ethical approval was not required.

Acknowledgements

The authors express their sincere gratitude to the scientific community and peer reviewers for their valuable feedback, which significantly improved the clarity, structure, and scientific rigor of this work. The authors also acknowledge the support of their affiliated institutions for providing the computational resources necessary for conducting the experiments.

Declaration of Generative AI in Scholarly Writing

All scientific content, ideas, methodology, and conclusions were developed and verified by the authors, who take full responsibility for the accuracy and integrity of the work.

Abbreviations

The following abbreviations are used in this manuscript:
FS – Feature Selection
GNN – Graph Neural Network

RL – Reinforcement Learning
CPU – Central Processing Unit
ML – Machine Learning

Appendix A\B\C…, with appendix tile
APPENDIX A FEATURE SELECTION FRAMEWORK FOUNDATIONS

The proposed framework integrates multiple feature selection methods into a unified architecture designed for intelligent software performance monitoring. The system operates on high-dimensional telemetry data and dynamically selects relevant feature subsets based on structural, statistical, and contextual characteristics of the monitored system. Unlike conventional approaches that rely on static feature selection techniques, the framework formulates feature selection as a multi-criteria optimization problem. It balances trade-offs between predictive accuracy, feature dimensionality, computational overhead, and monitoring cost. The architecture is modular and extensible, supporting deployment across monolithic, microservice, cloud-native, and serverless environments.

APPENDIX B FEATURE SELECTION DECISION

B.1 System Components
The framework consists of the following main modules:
- telemetry data collection module,
- preprocessing and normalization module,
- feature analysis module (statistical and structural),
- feature selection decision module,
- machine learning evaluation module,
- and monitoring and feedback interface.

The system operates in iterative cycles, allowing dynamic adaptation of selected feature subsets.

B.2 Feature Analysis Phase
Each telemetry dataset is analyzed to extract relevant characteristics, including:
- statistical properties (variance, correlation, distribution),
- temporal stability of features,
- structural dependencies between components (modeled as graphs),
- predictive contribution to target variables.

Feature importance is estimated using a combination of statistical metrics and model-based evaluation techniques.

B.3 Context Evaluation
The framework considers system-level context, including: workload intensity and variability, architectural type (monolithic, microservice, serverless), computational resource constraints, monitoring requirements (latency, accuracy). These factors influence the feature selection strategy.

B.4 Feature Selection Decision Phase
The decision module selects an optimal subset of features based on multi-criteria evaluation. The following strategies are applied:
- redundancy elimination using correlation analysis,
- topology-aware selection using graph-based models,
- multifactor scoring combining statistical and predictive metrics,
- adaptive selection using reinforcement learning.

The objective is to maximize predictive performance while minimizing feature dimensionality and computational cost.

B.5 Model Training and Evaluation
Selected features are used to train machine learning models for performance prediction or anomaly detection. Model performance is evaluated using accuracy, precision, recall, and stability metrics.

B.6 Adaptation and Optimization
The framework includes a feedback mechanism that continuously updates feature selection strategies based on observed performance. This includes: adjusting feature importance thresholds, updating model parameters, adapting to workload changes. Fallback mechanisms ensure stability in case of sudden system changes.

APPENDIX C EXPERIMENTAL SETUP AND VALIDATION

C.1 Experimental Environment

Experiments were conducted using large-scale telemetry datasets collected from 42 real-world applications. The dataset includes diverse system architectures and contains high-dimensional performance metrics.

C.2 Evaluation Scenarios

The framework was evaluated under multiple conditions: stable workloads, dynamically changing workloads, resource-constrained environments, large-scale distributed systems.

C.3 Evaluation Metrics

Performance evaluation includes: feature dimensionality reduction (%), predictive accuracy (%), feature stability, computational overhead, resource consumption efficiency. To ensure reliability, experiments were repeated multiple times, and statistical measures (mean, standard deviation, confidence intervals) were calculated.

C.4 Experimental Results Summary

The experimental results demonstrate that the proposed adaptive feature selection framework achieves:

- an average reduction in feature dimensionality of 37%,
- predictive accuracy above 95% across multiple models,
- an average improvement in dependency modeling accuracy of 21%,
- an average gain in feature relevance estimation of 18%,
- an average reduction in feature instability of 26% under dynamic workloads,
- and up to 42% decrease in monitoring-related resource usage in serverless environments,

as observed across repeated experiments under controlled workload variability and consistent evaluation conditions.

References

[1] Ghattas, M., Mora, A. M., & Odeh, S. (2025). A novel approach for evaluating web page performance based on machine learning algorithms and optimization algorithms. AI, 6(2). DOI: 10.3390/ai6020019

[2] Barbieri, M. C., Grisci, B. I., & Dorn, M. (2024). Analysis and comparison of feature selection methods towards performance and stability. Expert Systems with Applications, 249. DOI: 10.1016/j.eswa.2024.123667

[3] Epstein, E., Nallapareddy, N., & Ray, S. (2023). On the relationship between feature selection metrics and accuracy. Entropy, 25(12). DOI: 10.3390/e25121646

[4] Mostert, W., Malan, K. M., & Engelbrecht, A. P. (2021). A feature selection algorithm performance metric for comparative analysis. Algorithms, 14(3). DOI: 10.3390/a14030100

[5] Muthukrishnan, R., & Rohini, R. (2016). LASSO: A feature selection technique in predictive modeling for machine learning. Proceedings of ICACA. DOI: 10.1109/ICACA.2016.7887916

[6] Pawening, R., et al. (2016). Feature selection methods based on mutual information for classifying heterogeneous features. Jurnal Ilmu Komputer dan Informasi, 9(2). DOI: 10.21609/jiki.v9i2.384

[7] Onakpojeruo, E. P., & Sancar, N. (2024). A two-stage feature selection approach based on artificial bee colony and adaptive LASSO. Applied Mathematics, 4(4). DOI: 10.3390/appliedmath4040081

[8] Li, B., et al. (2021). Enjoy your observability: An industrial survey of microservice tracing and analysis. Empirical Software Engineering, 27(1). DOI: 10.1007/s10664-021-10063-9

[9] Colarusso, C., et al. (2024). A distributed tracing pipeline for improving locality awareness of microservices applications. Software: Practice and Experience, 54(6). DOI: 10.1002/spe.3317

[10] Panahandeh, M., et al. (2024). ServiceAnomaly: Anomaly detection in microservices using traces and profiling metrics. Journal of Systems and Software, 209. DOI: 10.1016/j.jss.2023.111917

[11] Zhu, Y., et al. (2024). MicroIRC: Root cause localization for microservice systems. Journal of Systems and Software, 216. DOI: 10.1016/j.jss.2024.112145

[12] Khemani, B., et al. (2024). A review of graph neural networks: Concepts, architectures, techniques, challenges, datasets, applications, and future directions. Journal of Big Data, 11. DOI: 10.1186/s40537-023-00876-4

[13] Dai, E., et al. (2024). A comprehensive survey on trustworthy graph neural networks. Machine Intelligence Research, 21(4). DOI: 10.1007/s11633-024-1510-8

[14] Van Riet, J., Malavolta, I., & Ghaleb, T. (2023). Optimize along the way: An industrial case study on web performance. Journal of Systems and Software, 198. DOI: 10.1016/j.jss.2022.111593

[15] Gomes, F., Rego, P., & Trinta, F. (2025). A systematic mapping study on observability of microservices-based applications: fundamentals, classifications, and challenges. Computing. DOI: 10.1007/s00607-025-01540-w

[16] Hammad, Y., Ahmad, A. A.-S., & Andras, P. (2025). An empirical study on the performance overhead of code instrumentation in containerised microservices. Journal of Systems and Software. DOI: 10.1016/j.jss.2025.112573

Authors' Profiles



Liubov Oleshchenko, Ph.D, Associate professor at the Computer Systems Software Department, Faculty of Program Systems and Applied Mathematics at National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine. Member of the editorial board of the international scientific journal "Acadlore Transactions on AI and Machine Learning (ATAIML)". Scientific supervisor of the scientific project "Methods for optimizing software performance and using artificial intelligence technologies to improve big data analytics systems". Scientific interests: AI, programming, computer networks, IoT, big data analytics, machine learning.



Zhengbing Hu, Prof., Deputy Director, International Center of Informatics and Computer Science, Faculty of Applied Mathematics, National Technical University of Ukraine "Kyiv Polytechnic Institute", Ukraine. Adjunct Professor, School of Computer Science, Hubei University of Technology, China. Visiting Prof., DSc Candidate in National Aviation University (Ukraine) from 2019. Research interests: computer science and technology applications, AI, network security, communications, data processing, cloud computing, education technology.



Andrii Dychka obtained his Ph.D in Software Engineering in 2024. He is currently an Assistant professor at the Computer Systems Software Department, Faculty of Program Systems and Applied Mathematics at the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ukraine. His research interests include issues related to programming, application software, software architecture, medical information systems, automatic identification, multicolor barcoding, digital medicine.

How to cite this paper

Liubov Oleshchenko, Zhengbing Hu, Andrii Dychka, "Feature Selection Methods for Intelligent Software Performance Monitoring Based on Machine Learning", International Journal of Engineering and Manufacturing (IJEM), Vol.16, No.3, pp.218-237, 2026. DOI:10.5815/ijem.2026.03.13