Computer Networks 274 (2026) 111872

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Detecting application transitions and identifying application types for intent-based network assurance: A machine learning perspective

John Violos [a,*], Fotios Voutsas [b], Christos Diou [c], Aris Leivadeas [a]

[a] École de Technologie Supérieure, 1100 Notre-Dame St W, Montreal, H3C1K3, Quebec, Canada
[b] Netdata Inc., 548 Market St #31942, San Francisco, CA, 94104-5401, USA
[c] Department Informatics & Telematics, Harokopio University of Athens, Omirou 9, Tavros, 177 78, Greece

ARTICLE INFO

Keywords:
Intent-based networks
Network assurance
Monitoring systems
Alert systems
Application transition detection
Application type identification

ABSTRACT

Intent-Based Networking (IBN) enables agile and policy-driven network management by translating high-level intents into concrete configurations and continuously validating their compliance. A critical limitation in current Intent-Based Network Assurance (IBNA) systems is the lack of real-time application-level awareness, particularly in dynamic edge environments where AI workloads frequently change. In this work, we address this limitation by introducing a lightweight, monitoring-driven pipeline that enables the detection of application transitions and identification of newly active application types on edge devices. In collaboration with Netdata engineers, we develop multimetric data collectors using Netdata, an open-source platform for real-time system and application monitoring. These collectors capture application-agnostic system metrics with minimal overhead, forming the foundation for real-time alerting and dynamic network adaptation. Our proposed pipeline transforms raw monitoring data into fixed-length vectorized multivariate time series. An undercomplete autoencoder is then used to detect changes in system behavior indicative of application transitions, followed by a Random Forest classifier that labels the newly active application based on its resource usage profile. To support reproducibility, we construct and publicly release the AIMED-2025 dataset, which includes monitoring data from seven MediaPipe-based edge AI applications and two idle states, all executed on a Raspberry Pi. Experimental evaluation demonstrates that our method achieves 100 % accuracy in both Application Transition Detection and Application Type Identification using only a three-second observation window. Furthermore, the system exhibits sub-second training times and millisecond-scale inference latency, making it suitable for real-time deployment on resource-constrained edge devices. Once an application change is detected and identified, the IBNA system can automatically alert network administrators and trigger dynamic reconfiguration of network resources to meet the specific performance, security, and connectivity requirements of the active application. By integrating application-level awareness into IBNA, this work advances the state of the art in intent-driven network management and enables more adaptive, efficient, and reliable operation of edge AI systems.

1. Introduction

Intent-Based Networking (IBN) is an emerging paradigm in which network users specify high-level goals or "intents" for the network, and the IBN system automatically implements these goals through underlying configurations [1]. Architecturally, a typical IBN framework includes stages for intent expression, translation into network policies, orchestration of devices, and closed-loop assurance [2]. In practice, the system translates abstract intents into concrete policies and pushes them to network devices, while a feedback loop continuously collects monitoring metrics to verify that the deployed network behavior matches the original intent [3]. This high-level abstraction and automation greatly enhances agility and reliability: by eliminating most manual configuration steps, IBN reduce human error and enable fast, policy-driven reconfiguration of the network. The Intent-Based Network Assurance (IBNA) component performs continuous monitoring and validation of the network state against the declared intents. It automatically detects deviations or "intent drift", which refer to any discrepancies between the intended behavior defined by the network's high-level goals and the actual operational state observed in real time. When such deviations are detected, the system triggers corrective remediation to restore compliance with the original intent.

Edge AI applications, executed directly on edge devices enables real-time inference, localized decision-making, and context-aware

* Corresponding author.
E-mail address: ioannis.violos@etsmtl.ca (J. Violos).

https://doi.org/10.1016/j.comnet.2025.111872
Received 7 August 2025; Received in revised form 28 October 2025; Accepted 14 November 2025
Available online 21 November 2025

operations [4]. These applications, ranging from object detection and gesture recognition to audio classification and pose estimation, introduce diverse and often stringent requirements on the underlying network infrastructure. For example, latency-sensitive tasks such as video-based face detection demand low end-to-end delay and consistent bandwidth, whereas applications like audio classification may prioritize jitter tolerance and sustained throughput [5]. Additionally, some applications require dedicated CPU/GPU resources, isolation through traffic segmentation, or the activation of specific communication protocols to ensure reliable operation. Because each application imposes distinct performance, security, and connectivity needs, the network must dynamically reconfigure itself whenever a new application is deployed [6]. This includes adjusting resource allocation, activating appropriate services and protocols, and ensuring that policies remain aligned with the real-time operational context.

In this work, we propose an IBNA-based framework to ensure the reliability and performance of Edge AI applications through a unified, monitoring-driven approach. The framework introduces an Application Transition Detection mechanism that continuously analyzes real-time monitoring metrics to identify transitions between AI applications running on edge devices. This is complemented by an Application Type Identification module that classifies the newly active application into a known category. Both mechanisms are integrated into a lightweight and efficient pipeline that uses monitoring data collected by multimetric data collectors developed in Netdata,[1] a scalable open-source platform for real-time system and application monitoring. By transforming raw monitoring data into actionable insights, the system enables automated IBNA responses. Once an application transition is detected and the application type is identified, the system can alert network administrators and trigger adaptive reconfiguration of network resources. This ensures that the network dynamically aligns with the specific performance and policy requirements of the active AI workload, highlighting the critical role of monitoring in delivering continuous assurance in intent-based networks.

To perform Application Transition Detection, we employ a novel detection mechanism, which identifies previously unseen patterns in monitoring data that deviate from the normal execution profile of a known application [7]. This is achieved using an one-class classification model trained solely on normal data to detect such deviations. This approach is well suited for identifying transitions between applications, as each application exhibits distinct resource utilization signatures in monitored system metrics. For Application Type Identification, we utilize a multiclass classification model trained to recognize and label each application based on its unique metric profile. To evaluate the effectiveness of our proposed methodology, we deployed seven edge AI applications and two idle states on an edge device and constructed the AI Monitoring at the Edge Dataset (AIMED-2025). Accordingly, the main contributions of this work can be summarized as follows:

- **End-to-End Pipeline for IBNA**: We develop and demonstrate a complete pipeline that retrieves and processes raw monitoring data to deliver IBNA through integrated Application Transition Detection and Application Type Identification mechanisms.
- **Novel Application Transition Detection via vectorized multivariate time series**: We design and evaluate machine learning models for Application Transition Detection, culminating in a novelty detection approach that leverages vectorized multivariate time series to accurately capture dynamic behavior shifts in edge application execution.
- **Public Release of Edge Monitoring Dataset**: We present and publicly release AIMED-2025, a dataset capturing monitoring data from diverse AI applications running on edge devices, to support research and benchmarking in the field of network assurance.

[1] https://www.netdata.cloud

The rest of the paper is organized as follows. Section 2 reviews the related work and background technologies on IBN and our proposed methodology. In Section 3, we present our overall methodology, detailing the core components of our approach. Section 4 describes the construction of the dataset for evaluating our framework. Section 5 provides the experimental evaluation of the data collection mechanisms, the Application Transition Detection model, and the Application Type Identification model. Finally, Section 6 concludes the paper.

2. Related work & background

IBN represent a transformative approach to network management, shifting from manual, device-level configuration to automated, goal-driven operation. In an IBN, users express their desired outcomes, referred to as intents, in a high-level, human-friendly manner [8]. The network then autonomously interprets these intents, deploys the appropriate configurations, and continuously ensures compliance. Rather than focusing on how a network should operate technically, IBN enable operators to define what the network should achieve, allowing the underlying system to determine the how [9]. This paradigm aims to increase operational efficiency, reduce human error, and enable more agile responses to changing requirements.

The two key characteristics that distinguish IBN are the automated implementation, which applies network configurations, expressed as intents, to devices without human intervention and closed-loop feedback, which allows the system to detect, analyze, and correct deviations automatically [3]. An IBN architecture is typically composed of five core components as we can see them in the Fig. 1: (1) Intent Profiling, where users express their goals in natural language or simplified interfaces; (2) Intent Translation, which interprets these goals into low-level policies; (3) Intent Resolution, which detects and manages conflicting or overlapping intents; (4) Intent Activation, which ensures safe and personalized deployment of the requested services; and (5) Intent Assurance, which continuously validates and adapts the network to maintain alignment with the user's intent over time [2]. Together, these components

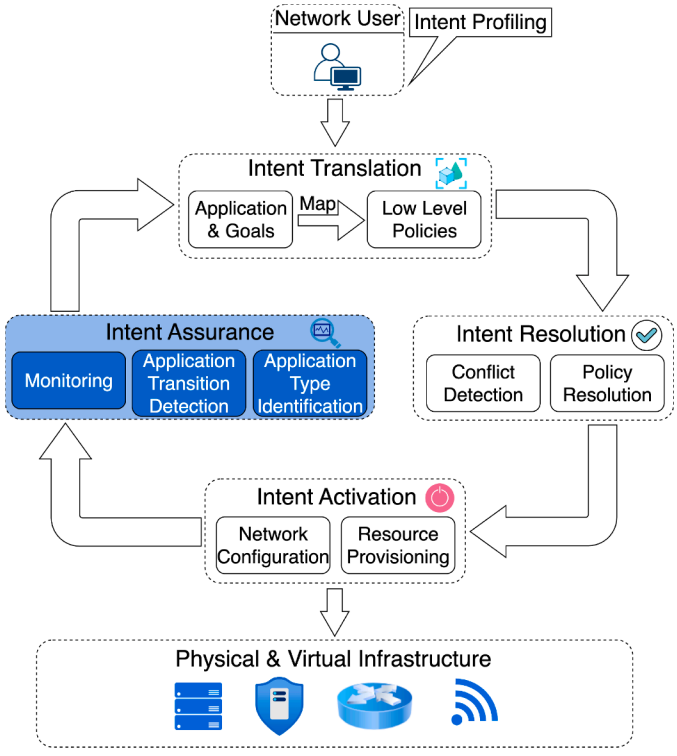

Fig. 1. The IBN components including the application transition mechanism and application type identification mechanisms.

enable IBN to support autonomous, intelligent, and user-centric network management.

To maintain alignment between the network's actual behavior and its intended goals, IBNA incorporates continuous monitoring mechanisms that observe network devices and traffic in real time [10]. This persistent verification process ensures that the deployed network state faithfully reflects the user's declared intents. When discrepancies arise such as unauthorized configuration changes, unexpected traffic patterns, or shifts in application behavior the system can autonomously detect these deviations and either suggest remediation steps or execute corrective actions automatically [11]. To facilitate such responsiveness, IBN can leverage monitoring platforms like Netdata, which provide performance metrics, system health visualization, and real-time alerts [12].

The recent advancements in IBNA focus on the integration of machine learning to enhance automation and adaptability. One notable research work is the use of AI-driven policies powered by Large Language Models, which are capable of understanding in-context requirements in order to assist the fulfillment and assurance of network intents [13]. In the context of Software-Defined Networks, a traffic prediction model has been proposed to proactively manage congestion by analyzing real-time network data and forecasting traffic patterns, enabling dynamic load balancing to maintain Quality of Service and intent compliance [14]. For data center environments, an incremental learning approach has been introduced to handle the evolving infrastructure by predicting key resource utilization metrics and allowing administrators to take timely corrective actions [15]. Additionally, a scalable solution using Neural Networks addresses the challenges posed by the growing customer base and big data demands, predicting bandwidth and other resource usage trends to support proactive network service assurance [16].

While IBNA encompasses key functions such as policy verification, which ensures that network configurations align with operational intents [11]; continuous compliance monitoring, which automatically checks adherence to organizational policies and regulatory mandates such as service-level agreements [17]; and conflict detection, which enables early identification and root cause analysis of network issues [18], the aspect of transition monitoring has been overlooked. Specifically, there is a research gap in examining how IBNA, when integrated with real-time monitoring tools, can be leveraged to detect application transitions. Exploring this capability is crucial not only for providing timely alerts to network administrators but also for enabling the system to reassess whether the network will continue to meet application-specific requirements like latency, throughput, and segmentation. Such an IBN would facilitate dynamic resource scheduling, activate the necessary ports, and employ the appropriate protocols based on the application's requirements.

The task of Application Transition Detection, has not been directly addressed in the existing literature. While some transition detection techniques have been developed, they are primarily focused on different domains such as detecting state transitions in intrusion detection systems [19], and do not address the specific requirements of application-level behavior monitoring. In contrast, application identification has been explored in areas like network security and traffic engineering, typically relying on features from packet header fields such as port numbers or through application-layer protocol decoding [20]. These approaches face notable limitations. Many applications use dynamic or non-standard ports, making header-based identification unreliable, and protocol decoding is often resource-intensive or infeasible in cases where protocols are encrypted or proprietary [21]. Furthermore, although artificial neural network architectures have been proposed to improve accuracy [22], they generally introduce high latency and still fall short of perfect prediction, which poses significant challenges for latency-sensitive IBN systems.

The design of the Application Type Identification component is straightforward: given the availability of multiple distinct application types, we approach the problem as a multiclass classification task. The choice of the specific classification model is guided by experimental evaluation, as presented in Section 5. In contrast, developing the Application Transition Detection mechanism involves exploring several methodological alternatives. Out-of-distribution detection aims to identify inputs that differ significantly from the distribution of the training data, signaling a shift in behavior or context [23]. Novelty detection focuses on uncovering previously unseen but valid inputs, assuming that the training data contains only examples of normal behavior [7]. Anomaly detection, meanwhile, seeks to detect rare or irregular patterns in the data [24]. Each of these approaches provides distinct capabilities for capturing transitions between applications in evolving network conditions and will be examined in the experimental evaluation.

Different concepts of intent-life cycle management have been proposed to enable autonomous and knowledge-driven network operations by integrating knowledge graph embeddings into the intent management framework [25]. This study defines a closed-loop intent life cycle encompassing intent expression, translation, validation, and mapping to their deployment. It introduces a dual closed-loop architecture in which the first loop manages intent representation and translation through knowledge graph based reasoning and inference, while the second loop handles deployment, monitoring, and optimization to ensure service-level compliance. Intents are modeled as resource description framework triples, allowing semantic understanding and intent completion through probabilistic reasoning in Gaussian embedding space. This knowledge-driven approach enables automated service orchestration and adaptive assurance, with compliance continuously verified against network dynamics using Simultaneous Perturbation Stochastic Approximation and Multiple Gradient Descent Algorithm.

While existing research has laid the groundwork for IBNA, it falls short in addressing the dynamic detection of application transitions in real-time environments. Our work closes this gap by introducing a unified framework that combines low-latency Application Transition Detection with precise Application Type Identification. Unlike prior techniques that rely on static heuristics, resource-intensive deep models, or unreliable packet-level features, our methodology leverages vectorized time series resource metrics and undercomplete autoencoders to deliver high-accuracy detection with minimal computational overhead. By further integrating this detection with a fast and robust multiclass classification model, our proposed methodology enables IBN to autonomously adapt to changing application contexts, facilitating policy realignment, resource reallocation, and continuous compliance. This represents a significant advancement in the state of the art, enabling networks to respond to evolving application requirements.

3. Proposed methodology

3.1. Overview

The proposed methodology for IBNA is grounded in the concepts of continuous monitoring and closed-loop automation, which are central to the operation of IBN. The placement of the monitoring, Application Transition Detection, and Application Type Identification mechanisms within the interaction flow of IBN components is illustrated in Fig. 1. The workflow of the proposed methodology is illustrated in Fig. 2. The process begins with edge devices running AI applications while data collectors gather key performance metrics in real time (Fig. 2a). These metrics are made accessible via APIs, can be used to generate health alerts, and are stored in a time series database for further analysis. After a fixed monitoring interval, the collected metrics, originally in the form of per-second feature arrays, are transformed into vectorized multivariate time series (Fig. 2b). This transformation concatenates feature metrics measured at sequential time steps to enable time-aware downstream processing. As will be discussed in the experimental evaluation, three one-second steps are sufficient for our analysis.

Once vectorized multivariate time series are collected over a sufficient period, approximately one hour based on our experiments, the system trains a novelty detection model to characterize the

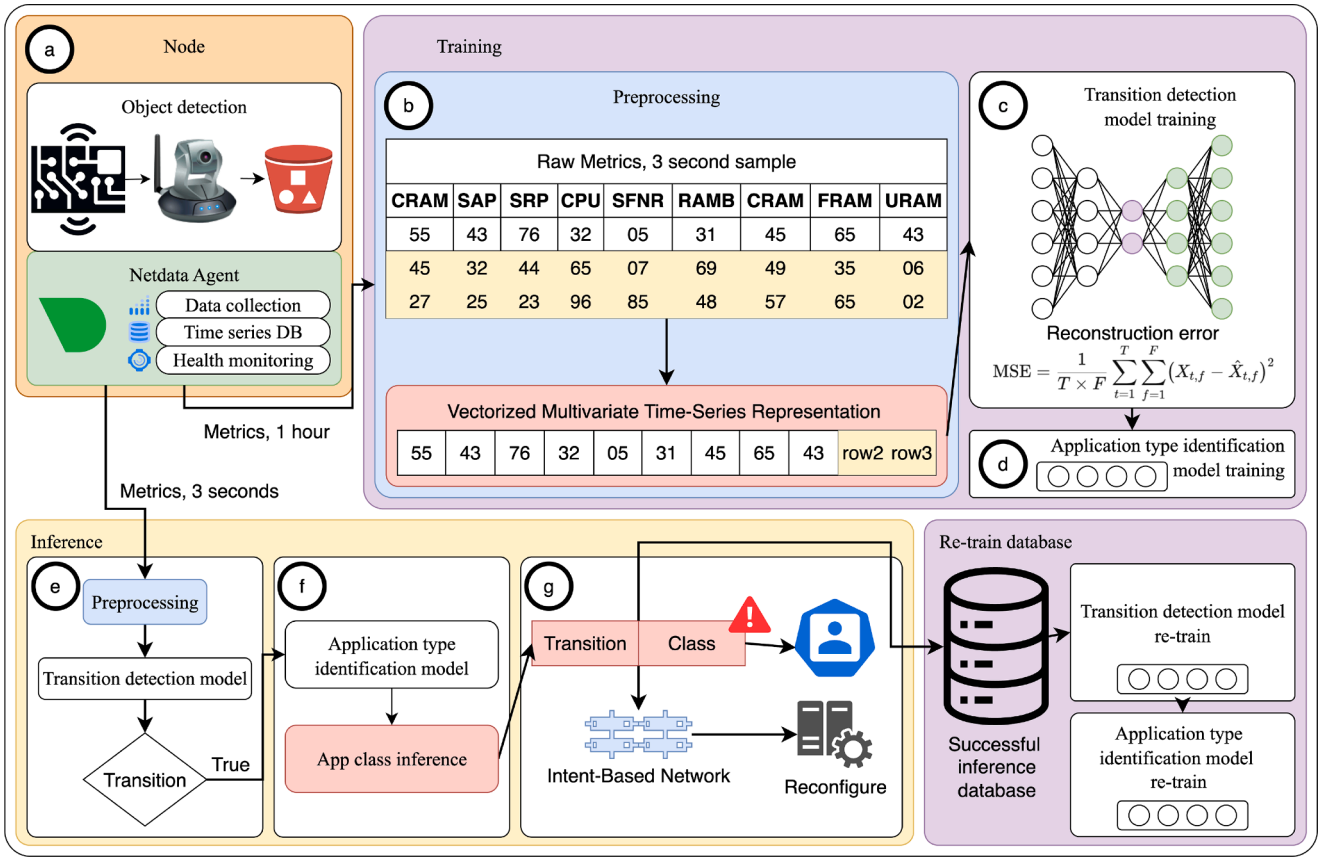

Fig. 2. Pipeline overview showing metric collection, transition detection, and type identification.

behavior of each running application on the edge device (Fig. 2c). This model enables the system to detect whether incoming metric data corresponds to the same application or indicates the execution of a new one. Furthermore, each time a new application type runs on the system, a multiclass classification model for Application Type Identification is incrementally trained. This model is updated using both the current instance and historical data stored in the time series database (Fig. 2d).

As new data continues to flow from the collectors, it is fed into the Application Transition Detection module, which determines whether an application switch has occurred (Fig. 2e). If a transition is detected, the updated Application Type Identification model is invoked to predict the type of the newly active application (Fig. 2f). This triggers a notification to the network administrator, providing information about the detected application transition and its predicted type. Based on this information, the IBN can automatically reconfigure itself to meet the specific requirements of the new application, thereby ensuring optimal network performance and policy compliance (Fig. 2g). In the following subsections we describe the key-modules of our methodology.

In the IBN architecture, illustrated in Fig. 1, the detection of a new application transition and its type serves as the initiating event that triggers a notification within the intent life cycle, propagating from the IBNA module to the Intent Translation, Resolution, and Activation components. The Intent Translation component interprets the detected application's goals and operational requirements, converting them into concrete low-level policies. These policies then guide the Intent Activation component, which is responsible for executing specific network adjustments such as slice reallocation, quality of service adaptation, and resource provisioning. While the present work focuses on the accurate detection and classification of application transitions, the design and optimization of the activation and execution mechanisms (e.g. policy enforcement algorithms, reallocation heuristics) are the responsibility of the other intent components and are beyond the scope of this study.

3.2. Data collectors

Data collectors are core components of network monitoring solutions that probe edge computing devices to gather critical performance and application metrics. The collectors operate continuously for each monitored device, application or service of interest. They are responsible for communicating directly with the target applications, extracting relevant metrics, and formatting the data in a way that can be parsed and visualized by the monitoring system's charting and alerting engine. This continuous stream of metrics enables IBNA models to make informed decisions about reconfiguring edge networks when necessary, while also providing network administrators with real-time insights into the operational status of the applications running on network devices. In our implementation, each data collector is realized as a lightweight job that interfaces with edge devices using standardized protocols such as SNMP, REST APIs, telemetry endpoints or even file reading. The collected metrics are then propagated to the central monitoring engine for further analysis and visualization.

3.2.1. Collector configuration

The collectors can be configured to monitor ports, commands, addresses, or APIs associated with the applications they monitor. Collector configuration can be performed either through the IBN system's auto-detection mechanism or by manually editing the relevant configuration files via the terminal or the monitoring tool's web-based user interface (UI). The UI simplifies the process, enabling operators to configure collectors without requiring SSH access to the monitored system. Furthermore, users can view and manage configurations across all nodes registered under their monitoring workspace. Upon submitting a

configuration, the system validates its structure and dispatches it to the corresponding node for immediate application.

Collectors can pose three types of behaviors upon attempting to collect metrics. Firstly, there are the default metrics the collector is looking for at Netdata startup. There is no user action needed, and metrics appear by default in the UI. The second behavior is for collectors that are specific to applications that expose their metrics on a port, file or some other source accessible from the system, without enforcing authentication. In that case, a Netdata component called "service discovery" polls for the existence of said applications in their known metrics outlets. If an instance is found, a collector job is invoked, and without the user's input or configuration, metrics begin to appear in the UI [26]. The third behavior is for applications that require authentication to access most metrics, like databases. Collectors will detect that a database is running on the system and monitor any allowed metrics, but the user must provide the necessary credentials or permissions to monitor the entirety of available metrics [27]. Additionally, the platform supports bulk operations, allowing the same configuration to be applied simultaneously across multiple applications within the user's workspace, streamlining large-scale management and ensuring consistency across environments.

3.2.2. API with application

Our data collectors provide a unified API interface that integrates with a wide range of applications for collecting system and service metrics. The collectors are developed in different programming languages depending on their complexity and required functionality. Bash is typically used for low-level tasks, while Python and Go are chosen for more sophisticated collectors. The decision is also influenced by language support within the system. For instance, Go collectors offer full integration and can be configured directly through the UI.

Additional collectors, aside from the default ones, are enabled whenever new services are detected or configured. The user can selectively disable any collector that he chooses to limit the observability of the monitoring system. Our proposed methodology uses default metrics that don't require further configuration or setup from the user. In a scenario where the user wants to provide his own selection of metrics to the model, accuracy might drop, but there is no realistic scenario where the user might want to limit the generic metrics we currently use [26,27].

Each collector operates through an underlying plugin that validates the configuration and initiates the data collection process. Once validation is complete, the collector executes a script that can interact with the system in several ways. It might invoke binary tools on the host machine, simulating terminal commands and parsing the output. It can also make HTTP requests to configurable endpoints and retrieve structured data in formats like JSON, XML, or CSV. In cases where services expose metrics through local files, the collector reads these as plain text and extracts the relevant values. This flexible design allows the collectors to support both modern and legacy environments effectively.

3.2.3. Parsing of the response

The response of a data collector is determined by the structure of the API request, which typically includes the target charts, a specific chart metric, the defined time windows, and the sampling intervals. The resulting data is returned in JSON format, which is then parsed and converted for storage in the monitoring system's time series database. In our implementation, each JSON response is aggregated in a CSV dataset which is then vectorized into fixed-length multivariate time series, where each column corresponds to a specific metric and timestamp. These vectors are subsequently used as input features to the Transition Detection and Application Type Identification learning pipelines.

3.2.4. Health alerts

To provide actionable insights beyond raw metrics, the monitoring tool supports health alerts based on configurable thresholds. The alerts and their thresholds are defined based on the intents specified by the network user. Internally, Netdata's alerting subsystem operates as a rule-based evaluation engine that continuously analyzes live metric streams. Each alert is defined by a declarative condition written in a domain-specific expression language capable of handling advanced logic, such as rate-of-change computations and rolling-window averages. Alert definitions are stored in configuration files and can be reloaded dynamically without restarting the monitoring agent. When evaluation thresholds are crossed, the system records a state transition (e.g., `CLEAR` →`WARNING` →`CRITICAL`) and enriches the resulting alert event with metadata including the triggering metric, timestamp, and node context. These alerts are displayed within the monitoring UI and can be sent through various notification channels. Additionally, users can configure webhooks, in cases such as when an application transition detected, to automatically respond to alerts, enabling a reactive approach to handling system issues and failures.

3.3. Monitored metrics & vectorized multivariate time series representation

To effectively support Application Transition Detection and Application Type Identification, we began with a broad set of candidate metrics spanning system performance, hardware utilization, application behavior, and network activity. This wide scope of metrics introduces the risk of including noisy or irrelevant features, which could negatively impact model performance. To address this, we collaborated with engineers from the Netdata team, leveraging their domain knowledge to refine the selection process. Their guidance helped us focus on metrics that are most indicative of application-level transitions and behaviors, while excluding those with low relevance or high variance unrelated to our task.

In parallel, we conducted a feature importance [28] analysis to further reduce the dimensionality of our input space. Our objective was to retain only the most impactful features in order to improve model efficiency, reduce training and inference times, and mitigate the risk of overfitting. The final set of monitored metrics, as recommended by Netdata experts and validated through our analysis, is summarized in Table 1. These metrics are application-generic, capturing low-level system behavior rather than application-specific logic, which ensures broad applicability across diverse edge AI workloads without requiring custom instrumentation. They concern Active and Running Processes on the system as well as CPU, File Descriptor and RAM utilization.

To construct the Vectorized Multivariate Time Series Representation, we concatenate the consecutive multivariate observations, where each observation corresponds to the system's monitored metrics captured at a fixed time interval, as detailed in Table 1. This approach encodes the temporal progression of the system's state over the sequential intervals into a single fixed-length vector, allowing models to learn patterns that evolve over time. We adopt a one-second timestep, as it represents the highest temporal resolution commonly supported by popular monitoring tools such as Netdata. Choosing a coarser timestep would introduce unnecessary delays in constructing the Vectorized Multivariate Time-Series Representations, delays that are undesirable given the need for detection mechanisms to respond promptly to recent system behavior.

3.4. Application transition detection

The Application Transition Detection module is designed to identify when a different application becomes active on a device, distinct from the one previously running, by analyzing temporal patterns in the edge computing network. This capability is essential for Intent-Based Network Assurance, as it enables the system to detect changes in application context that may require re-evaluating whether current network configurations continue to satisfy the declared intents. The Application Transition Detection module operates on vectorized multivariate time series derived from edge device monitoring metrics and determines whether the current monitored metrics correspond to the same application observed during the previous time period or indicates a transition to a new

Table 1
Monitored metrics.

Metric name	Unit	Description
System Active Processes	Processes/s	Total number of processes currently running, sleeping, or in other states
System Running Processes	Processes/s	Processes that are in the "runnable" state - either currently running on the CPU or waiting to run
System CPU utilization	Percent (%)	CPU usage across all cores. This metric refers to the user space CPU time, meaning time the CPU spends executing user-level code
System file NR	Files/s	Number of file descriptors in use
Committed RAM memory	Kb/s	Committed Memory, is the sum of all memory which has been allocated by processes
RAM Buffers	Kb/s	RAM used by the kernel to buffer block device operations such as writing to disk.
Cached RAM Memory	Kb/s	RAM used to cache files, which helps with performance by avoiding disk access.
Free RAM Memory	Kb/s	Total RAM not in use
Used RAM Memory	Kb/s	RAM actively used by running processes and kernel (excluding buffers/cache).

one. This is formulated as a novelty detection problem, where the objective is to detect previously unseen patterns based on a model trained only on data from known behavior.

Novelty detection involves learning the pattern of normal system activity and identifying inputs that deviate significantly from these expected behavior [29]. In our approach, we collect system metrics while a specific application is running, as detailed in Section 3.3, and use the resulting vectorized multivariate time series as positive samples of normal behavior.

A machine learning model, is trained on that data to capture the application's typical performance signature. During inference, the model computes a novelty score that reflects how far the current data deviates from the learned distribution. If this score exceeds a predefined threshold, the system flags the input as originating from a different, potentially new application. This threshold is selected during training by analyzing the distribution of novelty scores on known data and identifying a value that separates typical from atypical behavior.

We implement this detection mechanism using an autoencoder trained on vectorized multivariate time series generated from a known application. An autoencoder learns to encode the input data into a compressed latent representation and then decode it back to reconstruct the original input. When the input is similar to those seen during training, the reconstruction error remains low. In contrast, when the input originates from a different application or reflects a significant behavioral change, the reconstruction error increases, signaling to a potential application transition. For the reconstruction error, we use the Mean Squared Error computed across all features of the vectorized multivariate time series [30].

The implemented model follows an undercomplete autoencoder architecture, in which the latent representation is deliberately constrained to have fewer dimensions than the input [31]. This compression enforces the learning of compact feature embeddings that capture only the most informative aspects of the monitored behavior, thereby enhancing sensitivity to deviations from the training distribution. The encoder is composed of two fully connected layers: the first reduces the dimensionality to 50 % of the input size with a ReLU activation, while the second compresses further to 25 %, forming the latent bottleneck. The decoder mirrors this configuration to reconstruct the original input. In practice, this compact design results in a lightweight model that can perform real-time inference directly on the monitoring node, minimizing computational overhead while preserving high responsiveness to application transitions.

As it will be presented in the experimental evaluation Section 5, this approach effectively detects application transitions with high accuracy. The reconstruction error reliably increases when the system observes data from a new application, confirming the suitability of the undercomplete autoencoder for this task in the context of IBNA.

3.5. Application type identification

Once an Application Transition is detected, the Application Type Identification module is invoked to recognize the newly initiated application. This classification step is fundamental to enabling IBNA, as it allows the system to determine the nature of the active application and assess whether the current network configuration continues to satisfy the declared intent. By identifying the application type, the IBNA system can dynamically align resource provisioning, monitoring policies, and assurance mechanisms with the specific requirements encoded in the original intent, such as latency sensitivity, bandwidth demands, or isolation policies. The identification task is formulated as a multiclass, multivariate classification problem, where the output corresponds to one of a predefined set of known application types previously encountered in the edge environment and registered in the intent-aware configuration files. This bounded classification approach leverages historical data to improve prediction accuracy and ensures relevance to operational contexts defined by intent. The input features used for classification are derived from the same set of application-agnostic performance metrics described in Section 3.3, which are continuously collected on edge devices to support real-time decision-making in intent-driven networks.

To perform the classification, we employ a multiclass Random Forest model. This ensemble learning technique constructs a collection of decision trees, each trained on random subsets of the vectorized multivariate time series and their associated application type labels. Each decision tree partitions the feature space by selecting optimal threshold values for input metrics, guided by the Gini impurity criterion to maximize class separation [32]. By incorporating randomness both in data sampling and feature selection, the model achieves robustness to noise and reduces the risk of overfitting. During inference, a vectorized multivariate time series is propagated through all decision trees in the forest. Each tree issues an independent prediction, and the final application type is determined by a majority vote across the ensemble.

Each output of the Random Forest provides a confidence score indicating the likelihood that the running application belongs to one of the trained application classes. These confidence scores are derived from the normalized distribution of votes across all decision trees in the ensemble. When the system encounters an application that has been included in the training dataset, the confidence of the correct class is typically high, reflecting consistent agreement among the trees. Conversely, when the observed behavior corresponds to an application not present during training, the classifier exhibits uncertainty, and all class confidence scores remain low. In such cases, if the maximum confidence across all known classes falls below a predefined threshold, the system classifies the current application as unseen. This mechanism allows the Application Type Identification module not only to recognize known application types but also to detect novel ones.

In our implementation, the Random Forest classifier operates as a downstream module that is triggered immediately after the transition detection stage. The incoming vectorized time series segment [33], flagged as an application transition, is fed into the pre-trained model for type inference. The resulting class label can be consumed by the IBNA control plane to update network configurations or monitoring policies in real time. This modular pipeline design allows both models to function asynchronously, ensuring low-latency classification without disrupting ongoing metric collection or analysis.

3.6. The role of the application transition detection and type identification in network assurance and intent lifecycle

The IBN components are described in the related work and illustrated in Fig. 1. The role of application classification within intent assurance, as well as its relationship to the intent lifecycle, is depicted in Fig. 3. From the perspective of this lifecycle, the monitoring component of Intent Assurance plays a critical role by providing continuous feedback on the operational status of applications, devices, and network resources. The monitoring data collected during this phase forms the foundation for the proposed Application Transition Detection and Application Type Identification mechanisms.

When an application transition is detected, the system determines whether the newly active application corresponds to one of the known and previously learned types. If the transition involves a known application, the user is notified, and unless the user explicitly provides a new intent opposing the transition, the system implicitly interprets acceptance and resumes the intent lifecycle by reconfiguring the infrastructure and recirculating the IBN closed-loop. In this case, the IBN system continues with the stages of intent resolution, and activation to automatically adapt network configurations and resources to the requirements of the new application.

If the transition involves an unknown application, the system notifies the network users, who must provide additional information regarding the new application's low-level policies, intent resolution, and intent activation. These aspects are important for other IBN components but are beyond the scope of this work; interested readers are referred to [2] for further details. Furthermore, based on the monitoring data, new application models are trained after one hour of data collection, enabling the IBN framework to detect future application transitions and identify their types autonomously. This allows the system to manage subsequent occurrences of the same application type without manual intervention.

In contrast, if a network user objects to an application change, meaning the user explicitly rejects or refuses to run the new or modified application because it was initiated without their consent or against their will, this objection is interpreted as a new intent to terminate the application. This intent then follows the same resolution and activation processes to terminate the unauthorized or undesired application and restore the network to a compliant state.

By embedding the Application Transition Detection and Type Identification mechanisms into the Intent Assurance loop, the proposed framework enables the IBN system to dynamically interpret user preferences and adapt network behavior with minimal human intervention. This integration enhances network assurance, supports continuous intent alignment, and ensures that the IBN system remains responsive to evolving application contexts at the edge.

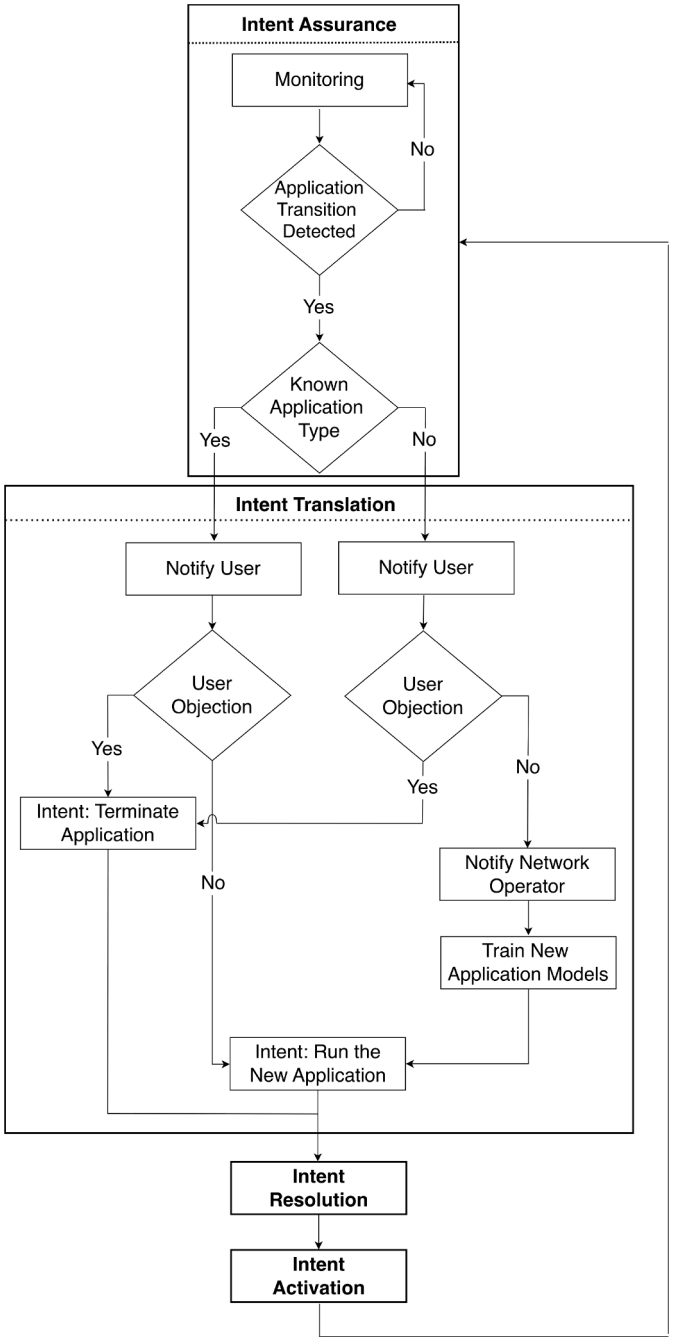

Fig. 3. Role of application classification within the intent lifecycle and associated processes.

4. Construction of the AIMED-2025 dataset

We constructed the dataset AIMED-2025, which is publicly available on GitHub [34]. We deployed Netdata data collectors on a Raspberry Pi running various AI applications to capture the metrics described in Section 3.3. Specifically, we monitored seven AI applications provided by MediaPipe [35]: Object Detection, Audio Classifier, Face Detector, Face Landmarker, Gesture Recognizer, Hand Landmarker, and Pose Landmarker, along with two idle workflows. These applications were selected because they represent common computational and sensory workloads encountered in real-world edge AI scenarios. In this section, we provide a brief overview of MediaPipe, the data collection process, and the applications included in our study.

MediaPipe[2] is an open-source framework developed by Google for building multimodal applied machine learning pipelines, with a strong focus on real-time perception tasks. Designed to run efficiently on edge devices MediaPipe provides a suite of pre-built solutions and customizable components for tasks including hand tracking, face detection, pose estimation, and object detection. It offers cross-platform support across Android, iOS, desktop, and web, enabling developers to deploy high-performance ML models with minimal latency. MediaPipe's modular architecture, built around a graph-based processing model, allows for efficient integration of computer vision and machine learning pipelines, making it well-suited for applications requiring real-time inference and interaction at the edge. The MediaPipe applications used in the construction of the dataset are described below.

- **Object detection**: An object detection application is a computer vision system that identifies and locates multiple objects within an image or video stream, typically by drawing bounding boxes around

[2] https://github.com/google-ai-edge/mediapipe

them and classifying each object. For this experiment, the camera was pointed at a television screen displaying a car race, and the application was tasked with detecting cars, humans, and other objects appearing in the video feed.

- **Audio classifier**: An audio classifier application is a machine learning system that analyzes audio input to identify and categorize sounds or acoustic events based on their characteristics. For this experiment, the microphone was placed in an environment where people were talking, animals could be heard and a television was also playing. The goal was to classify given sounds on a rolling timeframe.
- **Face detector**: A face detector application is a computer vision tool that identifies and locates human faces within images or video frames. For this experiment, the camera was placed in front of a working software engineer, where the subject was moving in and out of the frame, and the model was trying to identify a human face in the frame.
- **Face landmarker**: A face landmarker application detects and tracks key facial landmarks, such as eyes, nose, and mouth positions, to analyze facial geometry and expressions. For this experiment, the setup was the same as with the Face detector experiment, and the model was trying to identify and landmark face characteristics (eyes, nose, lips) in a live feed.
- **Gesture recognizer**: A gesture recognizer application interprets hand or body movements from visual input to identify specific gestures or actions. For this experiment, the process was similar with the Face landmarker example, only this time the subject was a hand palm, where it was changing gestures while typing on the keyboard and handling a mouse.
- **Hand landmarker**: A hand landmarker application detects and tracks key points on the hand to analyze its position, shape, and movements. For this experiment, the application was trying to identify the elements of a hand, fingers and finger-joints, in a live feed.
- **Pose landmarker**: A pose landmarker application detects and tracks key points on the human body to analyze posture and movement. For this experiment, we used a feed from a conference video, and was trying to track human body elements like the torso, arms, head, hips and legs of a subject moving through space.
- **Idle1**: An idle workflow refers to a state where the device is powered on but not running any active applications or processing workloads. For this experiment, we left the system idle, and captured the time series metrics.
- **Idle2**: A second idle workflow was created similarly with the Idle1 by leaving the system inactive and recording the time series data.

All applications were executed on the same hardware setup: a Raspberry Pi paired with a generic webcam that includes a built-in microphone. While some applications shared similar objectives, such as extracting landmarks or detecting objects from the video feed, others differed in modality, such as the audio classifier, which relied on audio input from the webcam's microphone rather than visual data. This mix of overlapping and distinct application purposes was intentional, allowing us to assess whether the differences in execution workflows could be accurately identified and classified using the methods employed in our experimental evaluation.

5. Experimental evaluation

We implemented the proposed methodology, as described in Section 3, and evaluated it using the datasets detailed in Section 4. Section 5.1 presents the evaluation of Netdata data collectors in a large-scale infrastructure, while Section 5.2 provides two snapshots illustrating the graphical representation of monitoring metrics during dataset construction. In Section 5.3, we present an experimental evaluation of the Application Transition Detection mechanism, comparing the performance and efficiency of the various machine learning models it employs. Section 5.4 presents a similar comparative evaluation for the model used to identify application types. Finally, Section 5.5 summarizes the key findings and insights gained from the experiments.

5.1. Evaluating collectors

Implementing the collectors as described in Section 3.2 we can monitor and process millions of metrics for large infrastructures, where thousands of metrics are auto-discovered per node (devices, virtual machines, applications), and there are thousands of nodes. In order to experimentally compare the performance of collectors inside the Netdata monitoring tool we compared them with Prometheus,[3] a well-established monitoring tool in the industry. In this comparison, we present the key differences in optimization and resource utilization related to collecting and handling large volumes of metrics on large systems, rather than focusing solely on the limited requirements of edge computing.

We tested 4.6 million metrics, on ingestion, hardware utilization, metric storage retention and API queries.[4] On CPU and Memory, Prometheus required 15 cores and 383 GiB of memory to handle the metrics, with regular freezes of ingestion, while the collectors of Netdata needed only 9 cores and 47 GiB. For retention, Prometheus was able to store 2 h of data at per-second granularity. With the same disk requirements, our solution retained 1.25 days worth of per-second data and using its automatic downsampling tiers it managed to keep historic data almost for 3 months. The disk IO of Prometheus was on average 147 MiB/s against Netdata's 4.7 MiB/s. On network usage, Prometheus used 515 Mbps, while our implementation used 448 Mbps.

Furthermore, on query performance on the API, our solution was 22 times faster than Prometheus in large queries, while also it preserved 100 % of the requested data, when Prometheus was having issues, having data loss due to scrape loss and achieving only 93.7 %. In conclusion, this comparison demonstrated that Netdata collectors offer significantly greater efficiency and scalability than the corresponding mechanisms of Prometheus, which is regarded as an established monitoring tool. In addition, Netdata collectors come with default configurations that enable them to capture large volumes of metrics and node relationships. In contrast, Prometheus requires advanced knowledge and extensive configuration, along with integration of additional tools, to achieve similar results.

5.2. Presenting monitoring metrics

During the construction of the AIMED-2025 dataset, we monitored the Raspberry Pi while it was idle and during the execution of the object detection application. Fig. 5 shows a snapshot of the metrics collected during the idle state, whereas Fig. 6 presents a snapshot of the metrics recorded during application execution.

The two figures consist of six Netdata charts each, containing one or more time series as "dimensions". At the top of the chart, the title is displayed along with some useful programmatic information, including the chart ID, which can be used for tasks such as API requests. The six charts shown (from top to bottom) represent: committed RAM memory, system active processes, system running processes, system CPU utilization, file descriptors, and system RAM usage. The RAM usage includes time series for free, used, cached RAM, and RAM buffers. The x-axis represents time steps, while the y-axis varies depending on the type of chart, with its title displayed vertically on the left side of each chart.

The timeseries are represented as colored lines, and are accompanied by the comprehensive Netdata UI to manipulate the view, like zooming in and out, highlighting areas, filtering the view from multiple sources to specific ones and more. Below each timeseries there is a matching colored vertical bar, along with the name of the dimension, current value

[3] https://prometheus.io

[4] https://www.netdata.cloud/blog/netdata-vs-prometheus-2025/

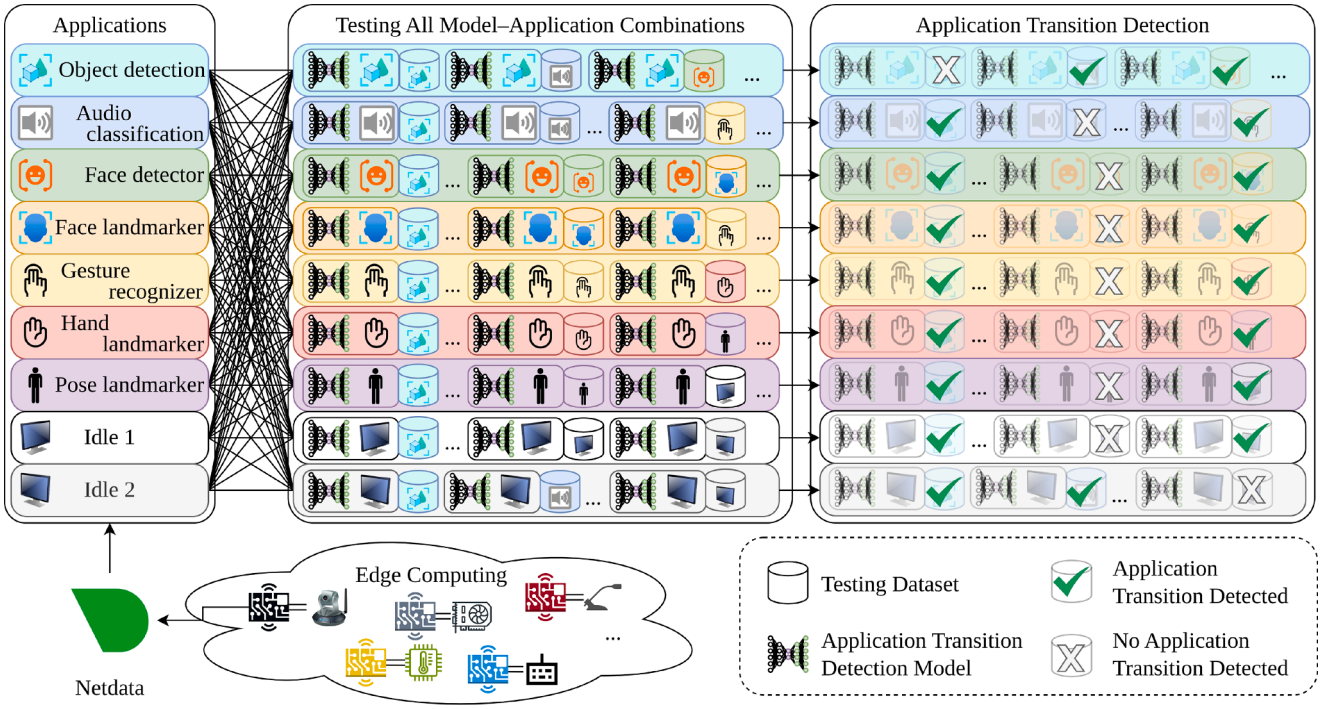

Fig. 4. Evaluation of each transition detection model across all application datasets.

and unit. In case there are multiple timeseries, like in the "System RAM" chart, there is an equal number of dimensions. This is preferred when the units are the same, and the timeseries can be visualized in a meaningful way using colors. As the charts have the same vertical scales, we can compare two workloads by taking two snapshots and previewing them side by side.

The comparison of the two figures reveals notable differences in system behavior between the two states. The committed memory usage of the object detection application increased significantly compared to the idle state. The number of active processes rose from approximately 2200 to 2900, while running processes experienced a pronounced spike, increasing from 3 to 73. Modest increases were also observed in CPU utilization and RAM usage, reflecting the heavier demands of the object detection workload.

The monitored metrics are leveraged by the Application Transition Detection mechanism. The novel detection model is trained using metrics collected during application runtime and can subsequently determine whether new incoming metrics correspond to the same application or indicate a transition to a different one, as demonstrated in the following subsection.

5.3. Testing models for application transition detection

5.3.1. Experimental evaluation protocol & evaluation metrics

To assess the performance of the proposed Application Transition Detection methods, we designed an experimental protocol involving individual training and cross-application testing. Specifically, we trained a separate machine learning model for each of the distinct applications and idle states described in Section 4. This resulted in a total of nine independently trained models, one per application or idle state. In our testing, each model occupied approximately 40 KB of space, which means that even for the extreme scenario of having 100 different known applications that could run on the same system, we would need around 4MB of space to store their corresponding autoencoders. The footprint of the models is very small, thus making it efficient to keep a model per application.

Following training, each model was evaluated using monitoring data corresponding to all seven application and two idle state scenarios. This approach produced a total of 81 (9×9) evaluations, capturing all possible combinations of training and testing application-state pairs. The full evaluation process is illustrated in Fig. 4. For instance, in the context of audio classification, a model trained exclusively on metrics from the audio application was tested not only on additional audio metrics but also on metrics from the other applications and idle states. This strategy enabled us to investigate the capabilities of each model trained in one application to detect the transition across any different application.

To evaluate performance, we used standard classification metrics: precision (Prec.), recall (Rec.), F1-score (F-1), and accuracy (Acc.). In addition, we included specificity (Spec.) to assess whether a detection model failed to recognize when both the training and testing data originated from the same application, a scenario that should ideally not be flagged as a transition. For efficiency evaluation, we measured the training time ($Time_{Train}$) of each model, the average inference time one sample ($Time_{Inf}$), and the average inference time of a batch of 100 samples ($Time_{BInf}$).

5.3.2. Outcomes

We conducted an experimental comparison of three distinct approaches for detecting application transitions: Out-of-distribution detection, Anomaly detection and Novelty detection. These approaches were selected based on our investigation of the machine learning literature, which revealed that they represent the main methodological categories applicable to identifying shifts or transitions in data behavior. For Out-of-distribution detection [36], we evaluated the Seasonal Ratio Scoring [23] using both 135-step and 10-step windows (SRS-135, SRS-10). For Anomaly detection[37], we tested autoencoders with 1-step (AE-1) [24], isolated forests [38] with 1-step and 3-steps (IF-1, IF-3), long short-term memory [39] with one-step and 3-steps (LSTM-1, LSTM-3) and the autoencoder with LSTM layers [40] with 10-steps (LSTM-AE-10). For Novelty detection we used autoencoders [41] with 3-steps (AE-3). Model names include a digit suffix indicating the length of the look-back window used [42]. In all experiments, each step corresponds to a

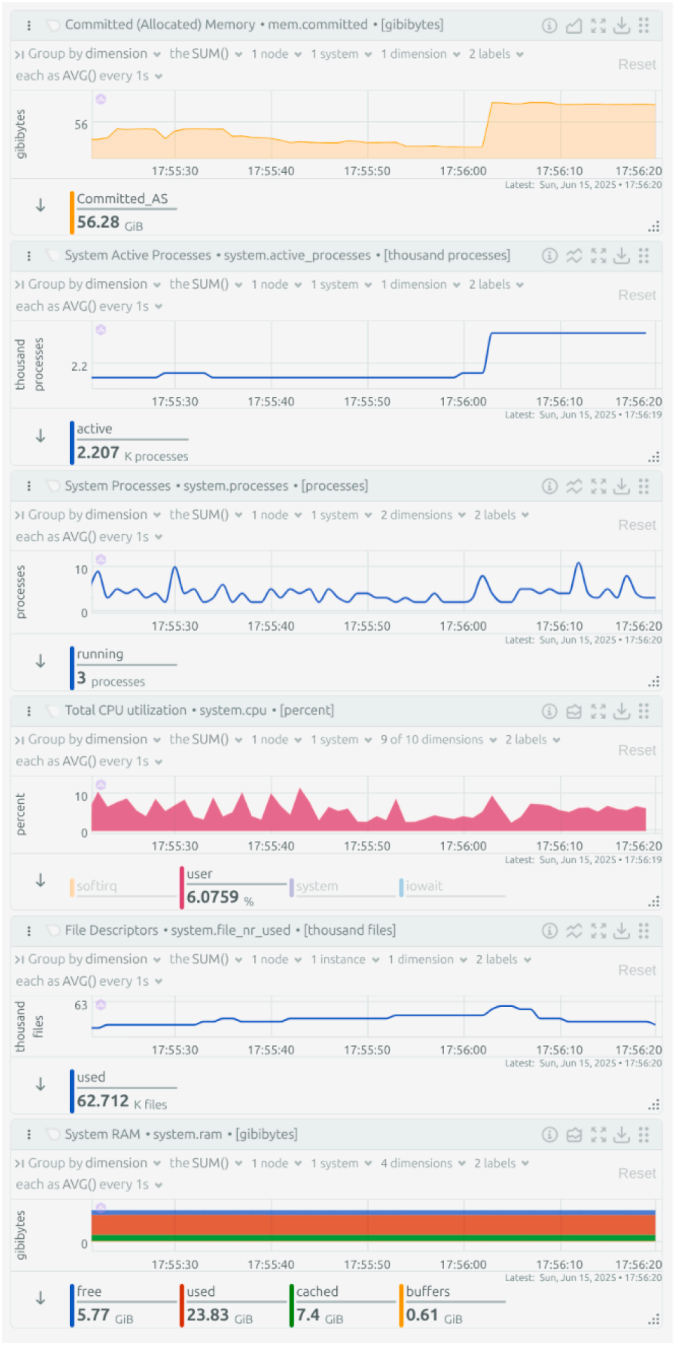

Fig. 5. Workload: idle state.

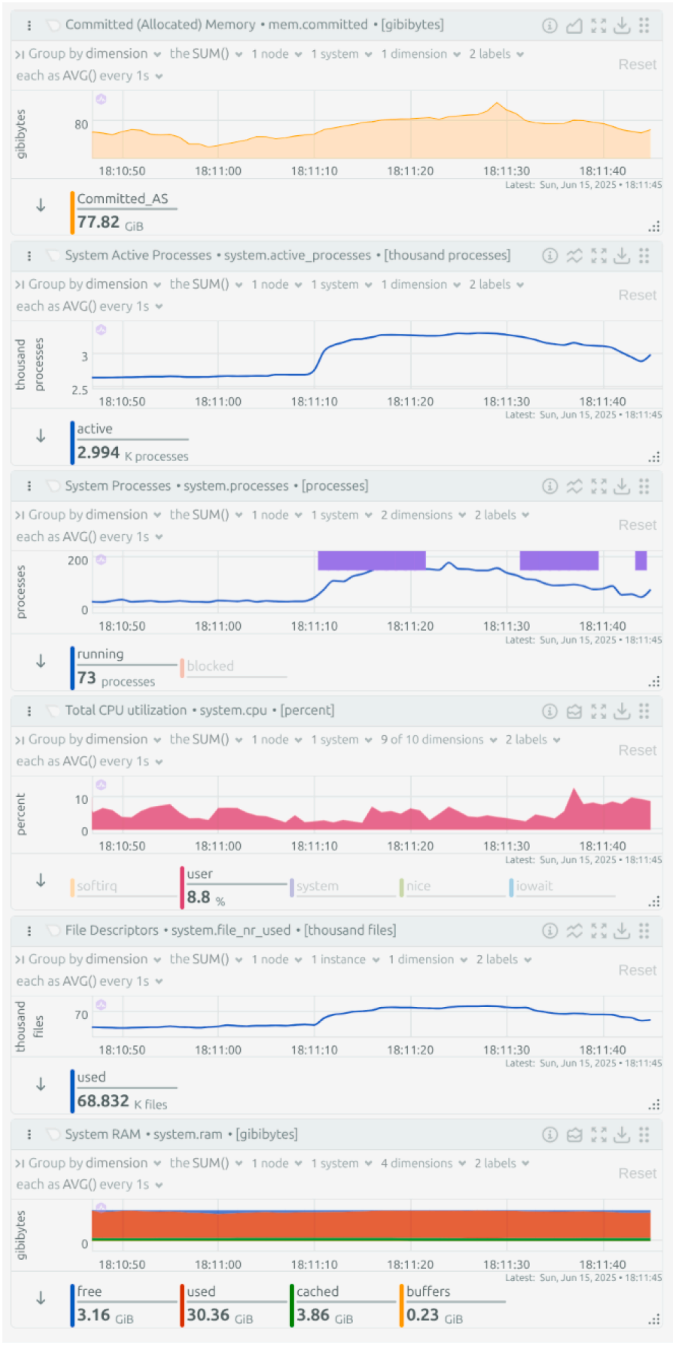

Fig. 6. Workload: object detection application.

one-second interval. The performance and efficiency results are summarized in Table 2.

The experimental results show that the AE-3 following the novelty detection approach and LSTM-3 following the anomaly detection, deliver the highest performance, achieving 100 % accuracy, F1-score, and specificity, while also maintaining low computational overhead during both training and inference. Both methods require approximately 3 s to generate the three-step vectorized multivariate time series. Furthermore, the AE-3 takes 0.032 s per single sample inference and 3.626 s for a batch of 100 samples, while the LSTM requires 0.034 s per sample and 3.815 s for batch inference. These outcomes indicate that autoencoders are the best option, as they are the most resource-efficient among the highest-performing models.

In terms of efficiency, IF-3 achieved the fastest training time at 0.105 s, although their performance metrics were slightly lower than those of AE-3. ARIMA provided the quickest inference times, with 0.001 s for a single sample and 0.152 s for a batch of 100 samples, but this came at the cost of significantly lower performance. Out-of-distribution detection methods intrinsically rely on long look-back windows to capture temporal dependencies, which is why we initially evaluated a 135-step window. This configuration yielded results with an F1-score of 0.957 and specificity of 0.789, indicating reasonable performance. However, this came at the cost of poor efficiency, as training took 81 s and inference averaged 40 s, resulting in a total of 175 s from data monitoring to prediction. This is far too slow for detecting application transitions, where near-instant response is required. To improve speed, we reduced the window to 10 steps, which achieved an F1-score

Table 2
Evaluation outcomes of application transition detection models.

Model	Prec.	Rec.	F-1	Acc.	Spec.	$Time_{Train}$	$Time_{Inf}$	$Time_{BInf}$
SRS-135	0.972	0.943	0.957	0.925	0.789	81.387	40.000	42.000
SRS-10	0.888	1	0.941	0.888	0	84.227	30.000	35.000
AE-1	0.920	0.985	0.951	0.913	0.454	6.335	0.083	8.842
IF-1	0.920	0.985	0.951	0.913	0.454	0.109	0.040	4.164
LSTM-1	0.909	1	0.952	0.913	0.363	7.247	0.066	7.233
ARIMA-1	0.864	1	0.927	0.864	0	2.576	0.114	11.866
AE-3	**1**	**1**	**1**	**1**	**1**	0.795	0.032	3.626
IF-3	0.972	1	0.985	0.975	0.818	**0.105**	0.019	2.130
LSTM-3	**1**	**1**	**1**	**1**	**1**	4.994	0.034	3.815
ARIMA-3	0.886	1	0.939	0.888	0.181	2.846	**0.001**	**0.152**
LSTM-AE-10	1	0.457	0.627	0.530	1	149.192	0.039	0.309

Table 3
Evaluation outcomes of application type identification models.

Model	Prec.	Rec.	F-1	Acc.	Spec.	$Time_{Train}$	$Time_{Inf}$	$Time_{BInf}$
RF-3	**1**	**1**	**1**	**1**	**1**	**0656**	**0004**	**0339**
LSTM-3	0314	0429	0333	0429	0905	4632	0346	3383
CNN-3	1	1	1	1	1	2015	0095	3329
RF-10	1	1	1	1	1	0338	0003	0335
LSTM-10	0314	0429	0333	0429	0905	3202	0289	3395
CNN-10	0786	0857	0810	0857	0976	1185	0089	3466

of 0.941 but resulted in a specificity of 0, meaning the model predicted almost all samples as positive. These findings show that the method is not only too slow for real-time detection but also ineffective for accurate detection.

5.4. Testing models for application type identification

5.4.1. Experimental evaluation protocol & evaluation metrics

To assess the applicability of machine learning models for Application Type Identification, we employed an experimental evaluation protocol in which each model was trained as a multiclass classifier using 70 % of the monitored metric sequences from the workload of the applications and the idle states of the dataset described in Section 4. Accordingly, the application types employed for both training and testing include Object Detector, Audio Classifier, Face Detector, Face Landmarker, Gesture Recognizer, Hand Landmarker, Pose Landmarker, and the two Idle States. The remaining 30 % of the data was used for testing, where sequential segments of monitored metrics were provided as input to evaluate model performance. For classification evaluation, we used the same performance and efficiency evaluation metrics described before.

To evaluate the performance of the proposed methodology on previously unseen applications, we conducted experiments involving two new workloads that were not part of the AIMED-2025 dataset. The first application was a lightweight Redis key-value store,[5] for which we generated workload traffic using the memtier benchmark performance testing tool.[6] The second application was an NGINX edge proxy,[7] for which we generated workload using the wrk HTTP benchmarking utility.[8] The monitoring procedure remained identical to that used for the AIMED-2025 dataset, collecting the same system and resource utilization metrics listed in Table 1. The data obtained from these two applications were excluded from model training and were used solely to assess the ability of the Application Type Identification model to recognize when the edge device is executing an application that has not been encountered before.

[5] https://pimylifeup.com/redis-docker/
[6] https://redis.io/docs/latest/operate/rs/clusters/optimize/memtier-benchmark/
[7] https://pimylifeup.com/docker-nginx-reverse-proxy/
[8] https://github.com/wg/wrk/

5.4.2. Outcomes

We conducted experiments using three well-established methods for multiclass classification: Random Forests [43] with 3-steps and 10-steps look back window (RF-3, RF-10), Long Short-Term Memory networks [44] (LSTM-3, LSTM-10), and Convolutional Neural Networks [45] (CNN-3, CNN-10). The results of these evaluations, covering both classification performance and computational efficiency, are summarized in Table 3.

The experimental results demonstrate that the RF-3, RF-10 and CNN-10, achieved 100 % accuracy in correctly identifying the application type. From an efficiency standpoint, the RF-3 outperformed the others, achieving sub-second training time and millisecond-level inference time. As a result, the Random Forest model with a 3-step window offers the best balance between accuracy and efficiency for the Application Type Identification task.

The LSTM model consistently performed the worst in both predictive performance and computational efficiency. Furthermore, both the LSTM and CNN models required significantly more time for training and inference, with inference times in the range of several tenths of a second. While the CNN-10 reached 100 percent accuracy, its longer training and inference times and the additional 7-s delay before producing an output make it less suitable for time-sensitive applications.

When executing applications that belong to one of the previously trained classes of the AIMED-2025 dataset, the Application Type Identification model consistently produces high confidence scores for the correct class, typically close to or exceeding 0.9. In contrast, when evaluating applications that were not included in the training set, the model exhibits significantly lower confidence levels across all known classes, with maximum scores remaining below 0.45. Table 4 presents the

Table 4
Class confidence of application type identification on unseen data.

Class	Redis	NGINX
Object detection	0,01	0,02
Audio classifier	0,37	0,4
Face detector	0,02	0
Face landmarker	0,12	0,04
Gesture recognizer	0	0
Hand landmarker	0,32	0,43
Pose landmarker	0,16	0,11

confidence values obtained for the unseen Redis and NGINX applications using the Random Forest trained on the application classes presented in Section 4. By establishing a confidence threshold of 0.45 to distinguish between known and unseen applications, the proposed methodology achieves 100 % accuracy in identifying whether a running application has been previously observed or represents a new, unseen application.

5.5. Discussion

An important design choice in our pipeline was to perform Application Transition Detection prior to initiating Application Type Identification. This sequence was chosen because the multiclass classification model used for type identification cannot reliably detect unseen application types. It can only classify inputs among the known categories it was trained on. Even when we experimented with using prediction confidence scores to infer whether an input belonged to an unknown class, the accuracy remained significantly lower compared to the proposed approach. By introducing a dedicated novelty detection mechanism to identify transitions first, we ensured that classification is only attempted when a new application is likely present. This design improves reliability while also avoiding unnecessary computations.

Given the importance of computational efficiency in IBNA systems operating at the edge, we deliberately avoided resource intensive deep learning models. The autoencoder used for transition detection is an undercomplete variant, chosen specifically for its simplicity and low resource demands. While out-of-distribution detection techniques might be theoretically suitable for identifying changes in application behavior, we found them to be too computationally expensive for real-time use and therefore excluded them early in the design phase. Our goal was to support lightweight, low-latency operations that align with the limited capacity of edge devices.

Intent-Based Network mechanisms, including monitoring and orchestration components, should impose minimal overhead, as the primary role of the network infrastructure is to support the operation of actual applications rather than interfere with the application performance. To demonstrate the lightweight nature of our approach, we evaluated the overhead introduced by the developed Netdata collectors and monitoring processes, comparing them with Prometheus, one of the most widely used monitoring tools. For the Application Transition Detection and Application Type Identification mechanisms, we measured the time required to train their models ($Time_{Train}$), to make a single prediction in the decentralized approach ($Time_{Inf}$), and to make one hundred predictions in the centralized approach ($Time_{BInf}$). The response times presented in Tables 2 and 3 indicate the minimal execution overhead of these processes. All time metrics were computed directly within the code using Python's time library, by capturing timestamps immediately before and after the corresponding events.

Regarding the scalability of the proposed methodology, it is important to note that, regardless of the number of devices or applications operating in an edge environment, the inference process of the application transition detection method can function in both decentralized and centralized modes. In a decentralized configuration, each edge device runs its own novelty detection mechanism for the application it hosts, meaning that the addition of new devices does not affect those already operating, and scalability concerns do not arise. In a centralized configuration, a single edge server can execute multiple novelty detection mechanisms, each corresponding to a specific device-application pair. Our experiments indicate that the execution time for one hundred observations remains low, suggesting that a single server can efficiently handle a large number of models. The same rationale applies to the training process, which can also be performed either in a decentralized manner on individual devices independently or centrally on an edge server serving all connected devices. Although training requires more time, it occurs infrequently, typically only when a new application is introduced. In highly dynamic scenarios with a large number of applications, scalability can be further improved by deploying additional edge servers and distributing the models among them for training and inference, since the novelty detection models operate independently of one another.

Another important design parameter was the number of time steps used in constructing the data input. Including more time steps can provide additional historical information, which may improve the model's performance. However, this decision would introduce a delay in shaping the input, which is undesirable in scenarios that require timely detection and adaptation. Our experiments showed that both the Application Transition Detection and Application Type Identification modules could achieve 100 percent accuracy using a three-second window. This finding indicates that our models can deliver rapid and accurate responses, which is essential for maintaining performance in dynamic edge environments.

In addition to inference speed, training time is also critical, especially because new applications require the system to train new models. Since this training must often take place at the edge, where computational resources are limited, models must be lightweight. The proposed approach meets this requirement. Furthermore, the prediction models can be deployed locally on each individual monitored device, or they can operate in a centralized manner on an orchestration node responsible for managing predictions across the entire edge network. In the centralized setup, the orchestrator processes inference in batches. In our experiments, we measured the time required to process batches for 100 devices using a Raspberry Pi. These results demonstrated that even on limited hardware, batch inference is feasible. On a more powerful orchestration machine, inference times would be significantly reduced.

The tracking of intents is carried out in an application-agnostic manner, relying on the highest degree of direct observation. Specifically, our methodology monitors the applications that users express through their intents independently of the application type and without requiring any intervention from the network user. This is achieved by observing the relevant metrics listed in Table 1, which allow the IBNA to infer the application intent fulfillment while maintaining application-agnostic monitoring.

The setup we followed ensured that the methodology can be applied on most standard monitoring solutions. Netdata is providing the most granular observability in the form of per-second metrics and is open source, so it was the preferred platform to capture the metrics. Our proposed solution can be integrated with other, except Netdata, open-source monitoring tools provided that their pipeline is adapted to their respective API and metric granularity.

Furthermore, the Netdata platform incorporates AI-driven insights, including local anomaly scoring and cross-node correlation analysis, to generate precise, actionable reports. Each report delivers a clear explanation of observed events, their underlying causes, and recommended next steps, presented in straightforward, accessible language. Building upon these capabilities, the Application Transition Detection and Application Type Identification mechanisms offer additional on-demand, detailed insights into the behavior and characteristics of running applications.

The proposed framework is designed to interoperate with monitoring frameworks beyond AI workloads by relying exclusively on generic, application-agnostic system metrics such as CPU utilization, RAM usage, and active processes which are universally available across all types of applications and operating systems. Since the transition detection and application identification models are trained on these low-level resource usage signatures rather than on AI-specific logic, the methodology can be directly applied to any workload, including web services, databases, or IoT applications, by simply retraining the models on the corresponding monitoring data. Furthermore, the pipeline is adaptable to other monitoring tools like Prometheus by aligning with their APIs and metric granularity, ensuring broad compatibility and making it a general-purpose solution for intent-based assurance in diverse edge environments.

The Application Transition Detection method requires one hour of monitoring data to train the undercomplete autoencoder. This duration was determined empirically by testing various time windows to identify the minimum monitoring period that maintains model accuracy. In scenarios where multiple rapid application transitions occur on the same device, the model will detect and report the first transition, but subsequent transitions within the same one-hour window may not be captured. This limitation is not critical, as the initial alert already informs the network operator, who can then investigate the cause of the application change. Nevertheless, extending the proposed methodology to handle multiple successive transitions represents a direction for future research, with techniques such as time series structural break detection offering a potential solution. Moreover, the current design assumes edge devices typically run a single isolated application, so another valuable avenue for future work is adapting the approach to environments where multiple applications may run in parallel, with one or more changing dynamically.

The proposed methodology can generalize across different types of edge workloads, hardware platforms, and more diverse AI applications. The Application Transition Detection and Application Type Identification models are developed using an undercomplete autoencoder and a random forest classifier, respectively, trained on generic monitoring metrics produced by the edge devices executing the applications. Since the metrics described in Table 1 are commonly available across all edge platforms and workloads, the same methodology can be applied to other devices and application types by repeating the data collection and model training process. In this work, we focus on AI applications due to their increasing adoption and significance in edge computing and smart environments [46]; however, future studies could further validate the method's robustness by extending it to different edge devices and non-AI workloads.

6. Conclusions

We have introduced a lightweight, end-to-end framework for IBNA that dynamically detects application transitions on edge devices and accurately identifies newly active workloads. By transforming high-frequency, multivariate Netdata metrics into fixed-length vectorized time series and applying an undercomplete autoencoder for low-latency novelty detection followed by a Random Forest classifier for multiclass application labeling, our pipeline achieves perfect accuracy and specificity while respecting strict edge constraints. The public release of the AIMED-2025 dataset, which captures seven MediaPipe edge AI applications and two idle states on a Raspberry Pi, provides a valuable resource for the network research community in studying and benchmarking application behavior at the edge.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used the ChatGPT in order to improve language and readability, with caution. After using the ChatGPT, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

CRediT authorship contribution statement

John Violos: Writing – review & editing, Writing – original draft, Visualization, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization; **Fotios Voutsas:** Writing – original draft, Visualization, Validation, Software, Resources, Data curation; **Christos Diou:** Supervision, Methodology; **Aris Leivadeas:** Project administration, Methodology, Conceptualization.

Data availability

The dataset used in this study will be made publicly available upon acceptance of the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Netdata engineers for their valuable collaboration and technical insights throughout the development of this work. We would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), grant No. RGPIN-2019-05250, for supporting in part our work. The authors would also like to thank the MPhil program in Computer Science and Informatics of Harokopio University of Athens https://mphil.dit.hua.gr/en/home/ for supporting this work.

References

[1] M. Falkner, J. Apostolopoulos, Intent-based networking for the enterprise: a modern network architecture, Commun. ACM 65 (11) (2022) 108–117. https://doi.org/10.1145/3538513
[2] A. Leivadeas, M. Falkner, A survey on intent-based networking, IEEE Commun. Surv. Tutor. 25 (1) (2023) 625–655. https://doi.org/10.1109/COMST.2022.3215919
[3] Y. Njah, A. Leivadeas, J. Violos, M. Falkner, et al., Toward intent-based network automation for smart environments: a healthcare 4.0 use case, IEEE Access 11 (2023) 136565–136576. https://doi.org/10.1109/ACCESS.2023.3338189
[4] X. Wang, Z. Tang, J. Guo, T. Meng, C. Wang, T. Wang, W. Jia, Empowering edge intelligence: a comprehensive survey on on-device AI models, ACM Comput. Surv. 57 (9) (2025) 228:1–228:39. https://doi.org/10.1145/3724420
[5] Q. Liang, W.A. Hanafy, A. Ali-Eldin, P. Shenoy, et al., Model-driven cluster resource management for AI workloads in edge clouds, ACM Trans. Auton. Adapt. Syst. 18 (1) (2023) 2:1–2:26. https://doi.org/10.1145/3582080
[6] K. Dzeparoska, N. Beigi-Mohammadi, A. Tizghadam, A. Leon-Garcia, et al., Towards a self-driving management system for the automated realization of intents, IEEE Access 9 (2021) 159882–159907. https://ieeexplore.ieee.org/abstract/document/9625012. https://doi.org/10.1109/ACCESS.2021.3129990
[7] M.A.F. Pimentel, D.A. Clifton, L. Clifton, L. Tarassenko, A review of novelty detection, Signal Process. 99 (2014) 215–249. https://doi.org/10.1016/j.sigpro.2013.12.026
[8] S. Minhas, R. Jaswal, A. Sharma, S. Singla, Revolutionizing networking: a comprehensive overview of intent-based networking, in: 2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP), 2024, pp. 463–468. https://doi.org/10.1109/INNOCOMP63224.2024.00081
[9] A.A. AlSamarneh, A.T. Al-Hammouri, O.Y. Al-Jarrah, Navigating intent-based networking: from user descriptions to deployable configurations, Neural Comput. Appl. (2025). https://doi.org/10.1007/s00521-025-11193-7
[10] A. Leivadeas, M. Falkner, Autonomous network assurance in intent based networking: vision and challenges, in: 2023 32nd International Conference on Computer Communications and Networks (ICCCN), 2023, pp. 1–10. ISSN: 2637–9430, https://doi.org/10.1109/ICCCN58024.2023.10230112
[11] Y. Song, C. Yang, J. Zhang, X. Mi, D. Niyato, et al., Full-life cycle intent-driven network verification: challenges and approaches, Netw. Mag. Glob. Internetworking 37 (5) (2023) 145–153. https://doi.org/10.1109/MNET.124.2200127
[12] F. Voutsas, J. Violos, A. Leivadeas, Mitigating alert fatigue in cloud monitoring systems: a machine learning perspective, Comput. Netw. 250 (2024) 110543. https://doi.org/10.1016/j.comnet.2024.110543
[13] K. Dzeparoska, A. Tizghadam, A. Leon-Garcia, Intent assurance using LLMs guided by intent drift, in: NOMS 2024-2024 IEEE Network Operations and Management Symposium, 2024, pp. 1–7. ISSN: 2374–9709, https://doi.org/10.1109/NOMS59830.2024.10575429
[14] M. Gharbaoui, B. Martini, D. Berardi, P. Castoldi, Towards intent assurance: a traffic prediction model for software-defined networks, in: 2025 28th Conference on Innovation in Clouds, Internet and Networks (ICIN), 2025, pp. 135–139. ISSN: 2472–8144, https://doi.org/10.1109/ICIN64016.2025.10942926
[15] S. Lévesque, X. Zheng, J. Violos, A. Leivadeas, M. Falkner, An incremental learning assurance approach for intent based networking enabled data centers, in: 2024 15th International Conference on Information, Intelligence, Systems & Applications (IISA), 2024, pp. 1–4. https://doi.org/10.1109/IISA62523.2024.10786660
[16] R. Hurtado, M. Torres, B. Pintado, A. Muñoz, Development of an intent-based network incorporating machine learning for service assurance of e-commerce online stores, in: E. Renault, P. Muhlethaler (Eds.), Machine Learning for Networking, Springer Nature Switzerland, Cham, 2023, pp. 12–23. https://doi.org/10.1007/978-3-031-36183-8_2

[17] Y. Sharma, D. Bhamare, N. Sastry, B. Javadi, R. Buyya, et al., SLA Management in intent-driven service management systems: a taxonomy and future directions, ACM Comput. Surv. 55 (13s) (2023) 292:1–292:38. https://doi.org/10.1145/3589339. https://doi.org/10.1145/3589339

[18] X. Zheng, View Profile, A. Leivadeas, View Profile, M. Falkner, View Profile, Intent based networking management with conflict detection and policy resolution in an enterprise network, Comput. Netw. 219 (C) (2022). Publisher: Elsevier North-Holland, Inc., https://doi.org/10.1016/j.comnet.2022.109457

[19] R.K. Pandey, T.K. Das, Anomaly detection in cyber-physical systems using actuator state transition model, Int. J. Inf. Technol. 17 (3) (2025) 1509–1521. https://doi.org/10.1007/s41870-024-02128-x

[20] L. Bernaille, R. Teixeira, K. Salamatian, et al., Early application identification, in: Proceedings of the 2006 ACM CoNEXT Conference, CoNEXT '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 1–12. https://doi.org/10.1145/1368436.1368445

[21] S. Zander, T. Nguyen, G. Armitage, et al., Automated traffic classification and application identification using machine learning, in: The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l, 2005, pp. 250–257. ISSN: 0742–1303, https://doi.org/10.1109/LCN.2005.35

[22] S. Rezaei, B. Kroencke, X. Liu, Large-scale mobile app identification using deep learning, IEEE Access 8 (2020) 348–362. https://doi.org/10.1109/ACCESS.2019.2962018

[23] T. Belkhouja, Y. Yan, J.R. Doppa, et al., Out-of-distribution detection in time-series domain: a novel seasonal ratio scoring approach, ACM Trans. Intell. Syst. Technol. 15 (1) (2023) 8:1–8:24. https://doi.org/10.1145/3630633

[24] A.A. Neloy, M. Turgeon, A comprehensive study of auto-encoders for anomaly detection: efficiency and trade-offs, Mach. Learn. Appl. 17 (2024) 100572. https://doi.org/10.1016/j.mlwa.2024.100572

[25] K. Mehmood, K. Kralevska, D. Palma, Knowledge-driven intent life-cycle management for cellular networks, IEEE Trans. Netw. Serv. Manag. 22 (5) (2025) 4806–4826. https://doi.org/10.1109/TNSM.2025.3579547

[26] Netdata, Memcached | Learn Netdata. Published: 2 months ago, https://learn.netdata.cloud/docs/collecting-metrics/databases/memcached.

[27] Netdata, MySQL | Learn Netdata. Last updated on Sep 25, 2025, https://learn.netdata.cloud/docs/collecting-metrics/databases/mysql.

[28] H. Mandler, B. Weigand, A review and benchmark of feature importance methods for neural networks, ACM Comput. Surv. 56 (12) (2024) 318:1–318:30. https://doi.org/10.1145/3679012

[29] K. Yang, S. Kpotufe, N. Feamster, et al., Feature Extraction for Novelty Detection in Network Traffic, 2021. https://doi.org/10.48550/arXiv.2006.16993

[30] T. Tziolas, K. Papageorgiou, T. Theodosiou, E. Papageorgiou, T. Mastos, A. Papadopoulos, et al., Autoencoders for anomaly detection in an industrial multivariate time series dataset, Eng. Proc. 18 (1) (2022) 23. Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, https://doi.org/10.3390/engproc2022018023

[31] M.A. Hussain, M. Saif-ur Rehman, C. Klaes, I. Iossifidis, Comparison of anomaly detection between statistical method and undercomplete autoencoder, in: Proceedings of the 5th International Conference on Big Data and Computing, ICBDC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 32–38. https://doi.org/10.1145/3404687.3404689

[32] M. Dilshad, B. Almas, N. Tariq, H.B. Jazri, G.N. Alwakid, J.S. Khan, P. Kumar, R. Kumar, IoV Cyber defense: advancing DDoS attack detection with gini index in tree models, in: 2024 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC), 2024, pp. 1–8. https://doi.org/10.1109/ETNCC63262.2024.10767505

[33] M. Su, W. Zhao, Y. Zhu, D. Zha, Y. Zhang, P. Xu, et al., Anomaly detection of vectorized time series on aircraft battery data, Expert Syst. Appl. 227 (2023) 120219. https://doi.org/10.1016/j.eswa.2023.120219

[34] F. Voutsas, Ancairon/intent-networks-app-transitions-types, 2025. original-date: 2025-08-04T08:22:32Z, https://github.com/Ancairon/intent-networks-app-transitions-types.

[35] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M.G. Yong, J. Lee, W.-T. Chang, W. Hua, M. Georg, M. Grundmann, et al., MediaPipe: A Framework for Building Perception Pipelines, 2019. https://doi.org/10.48550/arXiv.1906.08172

[36] J. Yang, K. Zhou, Y. Li, Z. Liu, Generalized out-of-distribution detection: a survey, Int. J. Comput. Vis. 132 (12) (2024) 5635–5662. https://doi.org/10.1007/s11263-024-02117-4

[37] S. Wang, J.F. Balarezo, S. Kandeepan, A. Al-Hourani, K.G. Chavez, B. Rubinstein, et al., Machine learning in network anomaly detection: a survey, IEEE Access 9 (2021) 152379–152396. https://doi.org/10.1109/ACCESS.2021.3126834

[38] W.S. Al Farizi, I. Hidayah, M.N. Rizal, Isolation forest based anomaly detection: a systematic literature review, in: 2021 8th International Conference on Information Technology, Computer and Electrical Engineering (ICITACEE), 2021, pp. 118–122. https://doi.org/10.1109/ICITACEE53184.2021.9617498

[39] S. Parsai, S. Mahajan, Anomaly detection using long short-term memory, in: 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), 2020, pp. 333–337. https://doi.org/10.1109/ICESC48915.2020.9155897

[40] Y. Wei, J. Jang-Jaccard, W. Xu, F. Sabrina, S. Camtepe, M. Boulic, et al., LSTM-Autoencoder-based anomaly detection for indoor air quality time-series data, IEEE Sens. J. 23 (4) (2023) 3787–3800. https://doi.org/10.1109/JSEN.2022.3230361

[41] F. Del Buono, F. Calabrese, A. Baraldi, M. Paganelli, F. Guerra, et al., Novelty detection with autoencoders for system health monitoring in industrial environments, Appl. Sci. 12 (10) (2022) 4931. Number: 10 Publisher: Multidisciplinary Digital Publishing Institute, https://doi.org/10.3390/app12104931

[42] M.-C. Lee, J.-C. Lin, E.G. Gran, et al., How far should we look back to achieve effective real-time time-series anomaly detection?, in: L. Barolli, I. Woungang, T. Enokido (Eds.), Advanced Information Networking and Applications, Springer International Publishing, Cham, 2021, pp. 136–148. https://doi.org/10.1007/978-3-030-75100-5_13

[43] A. Prinzie, D. Van den Poel, Random forests for multiclass classification: random MultiNomial logit, Expert Syst. Appl. 34 (3) (2008) 1721–1732. https://doi.org/10.1016/j.eswa.2007.01.029

[44] P.S. Muhuri, P. Chatterjee, X. Yuan, K. Roy, A. Esterline, et al., Using a long short-term memory recurrent neural network (LSTM-RNN) to classify network attacks, Information 11 (5) (2020) 243. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, https://doi.org/10.3390/info11050243

[45] S. Potluri, S. Ahmed, C. Diedrich, Convolutional neural networks for multi-class intrusion detection system, in: A. Groza, R. Prasath (Eds.), Mining Intelligence and Knowledge Exploration, Springer International Publishing, Cham, 2018, pp. 225–238. https://doi.org/10.1007/978-3-030-05918-7_20

[46] A. Bimpas, J. Violos, A. Leivadeas, I. Varlamis, Leveraging pervasive computing for ambient intelligence: a survey on recent advancements, applications and open challenges, Comput. Netw. 239 (2024) 110156. https://doi.org/10.1016/j.comnet.2023.110156