Expert Systems With Applications 315 (2026) 131705

Contents lists available at ScienceDirect

Expert Systems With Applications

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

Oraculum: A model for self-adaptive system optimization in smart environments

Darlan Noetzold[a,b,*], Valderi Reis Quietinho Leithardt[b,c], Juan Francisco de Paz Santana[b], Jorge Luis Victória Barbosa[a]

[a] University of Vale do Rio dos Sinos (UNISINOS), São Leopoldo, Brazil
[b] University of Salamanca, Expert Systems and Applications Laboratory (ESALAB), Salamanca, Spain
[c] Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal

A R T I C L E I N F O

Keywords:
Predictive analytics
Reinforcement learning
Real-time monitoring
Smart environments
Self-adaptive systems

A B S T R A C T

Smart environments require adaptive resource management to handle dynamic workloads and system variability. Traditional solutions, which often rely on static configurations or heuristic adjustments, may not maintain performance as conditions change. This work presents Oraculum, an adaptive model that integrates real-time monitoring, predictive analytics, and automated decision-making. Unlike previous architectures that apply reactive or rule-based adaptations, Oraculum incorporates predictive reinforcement learning (TD3) to anticipate environmental changes and optimize reconfiguration decisions proactively. The model applies data-driven methods to adjust system configurations dynamically, improving both resource allocation and service quality. Experimental results demonstrate that Oraculum significantly reduces Mean Adaptation Time (MAT) compared to existing self-adaptive models while achieving an adaptation accuracy of 97%, overhead of 2%, and maintaining system stability at 98%. These findings highlight the advantages of predictive control in addressing the challenges of dynamic workloads and resource constraints in smart environments, offering a practical approach for maintaining consistent performance.

1. Introduction

Smart environments, as defined by Steventon and Wright, refer to digitally augmented spaces in which computational systems support and enhance physical activities (Steventon & Wright, 2006). These environments integrate embedded devices and communication infrastructures to provide interactive and context-aware services. Applications span residential automation, industrial control, and urban infrastructure, with continuous data collection and processing enabling operational responsiveness and system-level optimization.

Self-adaptation in such systems frequently adopts the Monitor-Analyze-Plan-Execute over a shared Knowledge base (MAPE-K) control loop. This model structures autonomous behavior by enabling real-time state monitoring, data-driven analysis, adaptive planning, and runtime actuation (Fang et al., 2025; Pinthurat et al., 2024). The knowledge component centralizes the contextual information and decision policies used throughout the loop. Various implementations apply this model to domains such as robotics, where graph-based abstractions like Deep State Representation (DSR) improve coordination and reasoning (Martinez et al., 2022). The inclusion of artificial intelligence (AI) techniques-including supervised learning, unsupervised learning, and reinforcement learning (RL)-into the MAPE-K structure supports data-informed decision-making and predictive adaptation (Wang et al., 2025).

Despite the adoption of MAPE-K and AI in smart environments, several limitations persist. Existing models rarely support generic, discretizable metrics monitoring, restricting flexibility in heterogeneous settings (Aguayo et al., 2024; Zhang et al., 2025c). In many cases, only partial implementations of the MAPE-K loop are observed, resulting in fragmented behavior and latency during the adaptation process (Zhao & Guo, 2024). These constraints hinder the responsiveness and effectiveness of adaptive mechanisms.

Furthermore, the heterogeneous and dynamic nature of IoT devices complicates interoperability and resource coordination (Shin et al., 2023). Adaptation strategies often fail to scale or generalize across domains, contributing to increased system complexity and degraded

* Corresponding author.
E-mail addresses: darlannoetzold@usal.es (D. Noetzold), valderi.leithardt@iscte-iul.pt (V.R.Q. Leithardt), fcofds@usal.es (J.F. deP. Santana), jbarbosa@unisinos.br (J.L.V. Barbosa).

https://doi.org/10.1016/j.eswa.2026.131705
Received 30 July 2025; Received in revised form 23 December 2025; Accepted 13 February 2026
Available online 16 February 2026

performance (Alfonso et al., 2021; Yang et al., 2025). Addressing these issues requires a modular and extensible architecture capable of operating across diverse configurations and workload conditions.

This work introduces a self-adaptive model named Oraculum, which integrates complete MAPE-K functionality with modular AI-based adaptation components. Unlike existing approaches that rely on handcrafted adaptation rules or domain-specific controllers, Oraculum leverages a unified learning-based strategy that combines supervised, unsupervised, and reinforcement learning to support runtime decision-making in heterogeneous and dynamic contexts. Its ontology-driven structure ensures consistent metric classification and facilitates cross-domain extensibility, enabling the system to operate without manual reconfiguration.

Building on this overall design, the main contributions of this work can be summarized as follows:

- A parameterized monitoring mechanism that supports any discretizable metric, providing flexibility and extensibility across diverse use cases with minimal configuration effort.
- A fully integrated MAPE-K loop with synchronized modules for monitoring, analysis, planning, execution, and knowledge representation. The architecture allows real-time knowledge exchange among modules, enhancing responsiveness and adaptability.
- Predictive modeling for proactive adaptation through real-time metric forecasting, enabling the anticipation of future anomalies and allowing the RL agent to reason ahead of events. This capability significantly reduces the Mean Adaptation Time (MAT) to near zero, maintaining Adaptation Accuracy (AA), Adaptation Overhead (AO), and Stability above state-of-the-art levels.
- Comprehensive evaluation under representative IoT workloads, demonstrating that Oraculum achieves robust adaptation behavior, improving stability, accuracy, and responsiveness, and confirming its scalability and reliability for deployment in real-world dynamic environments.

The remainder of this paper is organized as follows. Section 2 reviews related work in self-adaptive systems and smart environment monitoring. Section 3 presents the architectural design of the proposed model. Section 4 discusses implementation aspects and deployment considerations. Section 5 reports on the experimental setup and evaluation results. Section 6 examines limitations and design implications. Section 7 concludes the work and outlines directions for future research.

2. Related work

Table 1 presents a comparison of self-adaptive architectures based on selected performance metrics. The metric MAT represents the average time that a system requires to respond to environmental changes. The metric AA quantifies the proportion of adaptation actions that align with expected results. The metric AO refers to the additional computational or resource usage introduced by the adaptation process. Stability reflects the system's performance consistency after adaptation events.

The table also specifies the phases of the MAPE loop (Monitor, Analyze, Plan, Execute) addressed by each model, the categories of AI techniques applied, including supervised learning, unsupervised learning, and reinforcement learning, and the monitored metrics relevant to system behavior. These elements provide a comparative view of how different architectures incorporate adaptation strategies. Definitions and further discussion of these metrics appear in Section 5 (Validation).

The ten architectures listed in Table 1 were identified through a systematic literature review based on the guidelines proposed by Petersen et al. (2008). The process involved an initial search across major scientific databases, followed by the evaluation of 133 studies. Filtering was performed using predefined inclusion and exclusion criteria to retain only those aligned with the study's scope. Among the analyzed studies, ten were selected for comparison due to their relevance in terms of performance metrics, adaptive mechanisms, and the use of AI techniques in intelligent IoT environments. These works provide the foundation for the comparative analysis presented in this section.

It is important to note, however, that the performance values reported in Table 1 were extracted directly from the original studies. As such, they were obtained under heterogeneous conditions, using different datasets, evaluation scenarios, and simulation tools. The lack of standardization in experimental setups limits the direct comparability of the metrics presented. Furthermore, the methodologies used to calculate MAT, AA, AO, and Stability may vary between works and are not always fully disclosed.

To distinguish reproduced results from values reported in the literature, this study relies on a common evaluation framework. *Oraculum* reports MAT, AA, AO, and Stability computed in the SHIELD environment, a unified testbed that executes all experiments under controlled and repeatable conditions, and uses a parameterizable, ontology-backed metric set for consistent monitoring. The underlying ontology organizes runtime metrics into semantic groups and supports systematic configuration of what is measured in each scenario. For validation in this study, the ontology excludes the *Energy* and *Security* groups and monitors four groups: *Hardware* (CPU Usage, Memory Usage, GPU Usage, Storage Utilization), *Network* (Latency, Data Transfer Rate, Packet Loss), *Software* (Response Latency, Throughput, Garbage Collection Time), and *SLA* (Availability, Resilience, Stability).

Table 1 includes MAT, AA, AO, Stability, and monitored-metric information for comparative architectures, but these values come directly from the respective publications and reflect their own heterogeneous configurations. For example, some works monitor only *Energy* and *Traffic Load*, others focus on *QoS* or *IoT Traffic*, while others restrict monitoring to *Throughput*, *Latency*-related metrics, or application-level quality measures such as *Image Quality*, *Convergence*, or *Visual Quality*. Consequently, only Oraculum's MAT, AA, AO, and Stability values arise from a unified metric configuration in SHIELD. All other entries in Table 1 act as baselines reported in the literature, each tied to its original metric scope and experimental setting.

Sah et al. (2022) proposed the Aggressive Scheduling Medium Access Control (AS-MAC) protocol to optimize energy consumption and data delivery in Industrial Internet of Things (IIoT) networks. The protocol applies scheduling mechanisms to coordinate sensor node activity, forming a backbone for efficient data collection. Simulations reported improvements in packet delivery rate and energy usage compared to traditional Time Division Multiple Access (TDMA) schemes. However, the study did not evaluate AA, reported an AO of 11%, and a network stability of 74%, suggesting limitations under dynamic operating conditions. Moreover, AS-MAC focuses on a single-layer MAC control strategy and it does not incorporate learning-based predictive mechanisms or ontology-driven context modeling, which restricts its ability to generalize across heterogeneous smart-environment workloads.

Velrajan and Sharmila (2023) introduced a Quality of Service (QoS)-aware service migration method for Multi-access Edge Computing (MEC) environments using a closed-loop particle swarm optimization approach (CLA-PSO). The model considers system load, application characteristics, and QoS constraints to enable proactive service migration. CLA-PSO reduced SLA violations compared to baseline methods and achieved 94% AA with a MAT of 1.8 s. The study did not assess AO, and the reported stability of 79% indicates potential performance variability during migration events. In addition, the optimization process is tailored to MEC service placement and does not provide a general MAPE-K architecture with unified metric representation or semantic reasoning, which limits its applicability beyond the specific migration scenario.

Etemadi et al. (2021) presented an auto-scaling mechanism for IoT applications in fog computing using deep learning for workload-aware resource allocation. The model achieved an AA of 91% with an average adaptation time of 1.2 s. However, AO was not evaluated, and stability was measured at 84%, suggesting susceptibility to workload fluctuations.

Table 1
Comparison of self-adaptive architectures.

Architecture	MAT (s)	AA (%)	AO (%)	Stability (%)	MAPE Cycle	AI Type	Metrics Monitored
Etemadi et al. (2021)	1.2	91	–	84	M,A	Supervised	CPU, Resources
Yang et al. (2021)	2.5	90	10	85	M,A,P	Supervised	Anomaly Detection
Liu et al. (2021)	2.7	87	12	82	A,P	Unsupervised	Anomaly Detection, Data Compression
Hameed et al. (2021)	2.8	–	14	75	M,A,P	Supervised	Throughput
Sah et al. (2022)	3.5	–	11	74	M,A,E	None	Energy, Traffic Load
Cen and Li (2022)	1.15	89	7	87	A,P	RL	Delay, Resource Allocation
Tam et al. (2022)	3.2	90	13	80	M,A,P,E	RL	Resource, Delay, Priority
Velrajan and Sharmila (2023)	1.8	94	–	79	M,A,P,E	RL	QoS, SLA
Samarakoon et al. (2023)	1.5	93	6	90	M,A,P,E	None	Latency, Bandwidth, Jitter
Priya et al. (2024)	1.9	92	7	89	M,A,P	Supervised	IoT Traffic
Wang et al. (2024)	2.1	93	6	88	M,A,P	RL	Convergence, Training Loss
Liu et al. (2024)	3.3	92	7	89	A,P	Supervised	Visual Quality, Fusion Distortion
Jamshidi et al. (2025)	–	92	–	–	M,A,P,E	DRL	Security, Energy Efficiency
Zhang et al. (2025a)	2.9	95	5	91	M,A	Supervised	Image Quality, Noise
Oraculum (Proposed)	0.05	94	4	98	M,A,P,E	Supervised, Unsupervised, RL	Parametric (any)

Cen and Li (2022) applied Deep Reinforcement Learning (DRL) to model resource allocation in cloud-edge collaborative computing as a Markov decision process. Using an enhanced Deep Q-Network (DQN), the system obtained 89% AA, 1.15 s MAT, and a 7% AO. Stability reached 87%, though the work emphasized delay minimization without fully addressing other adaptation metrics.

Yang et al. (2021) proposed a runtime anomaly detection algorithm selection service for IoT data streams based on Tsfresh feature extraction and a genetic algorithm. The system dynamically configures detection models to address input variability. The approach achieved 90% AA, 2.5 s MAT, 10% AO, and 85% stability. Longer adaptation times and moderate overhead may constrain its real-time deployment potential.

Liu et al. (2021) addressed anomaly detection in IIoT using Isolation Forest combined with compression techniques to minimize network latency. The model attained 87% AA, 2.7 s adaptation time, 12% AO, and 82% stability. The results suggest constraints in maintaining consistent behavior under variable network conditions.

Priya et al. (2024) developed a predictive optimization model using recurrent neural networks (RNNs) with long short-term memory (LSTM) to manage IoT traffic patterns. The method reached 92% AA, 1.9 s adaptation time, 7% overhead, and 89% stability. The study emphasized forecasting rather than runtime adaptation control.

Samarakoon et al. (2023) proposed a Kubernetes-based model for self-healing and adaptive management in IoT-edge infrastructure. The system integrated device performance data to trigger adaptation strategies. Results included 93% AA, 1.5 s MAT, 6% AO, and 90% stability. However, the reliance on Kubernetes introduces potential overhead in resource-constrained environments.

Hameed et al. (2021) applied regression-based machine learning to estimate throughput in a real-world IoT testbed composed of smart building applications. While specific accuracy metrics were not reported, the approach demonstrated a MAT of 2.8 s, AO of 14%, and stability of 75%, indicating sensitivity to variable traffic profiles.

Tam et al. (2022) investigated resource management for service function chaining (SFC) in IoT services using a priority-aware DRL mechanism. The model achieved 90% AA, 3.2 s adaptation time, 13% overhead, and 80% stability. The relatively long response time and computational overhead may reduce its suitability for time-critical operations.

Zhang et al. (2025a) proposed the Multi-scale Adaptive Residual Cold Diffusion (MarCoDiff) model for low-dose CT (LDCT) image denoising tasks. MarCoDiff employs cosine mean-preserving degradation operators and scheduling strategies to simulate the physical degradation process and accelerate sampling. The model incorporates adaptive modules to adjust the fusion of frequency components and reconstruct details at multiple scales, as well as a hybrid-dilated convolution context block to capture contextual information without further resolution loss. Experimental results demonstrated that MarCoDiff outperforms previous methods in both image quality and generalization under varying noise levels, highlighting its robustness and adaptability in medical imaging scenarios.

Wang et al. (2024) introduced AdaGC, a novel adaptive optimization algorithm with gradient bias correction for neural network training. AdaGC dynamically adjusts the iterative direction and step size based on gradient variation, promoting faster and more efficient convergence in both convex and non-convex problems. The algorithm was validated through extensive experiments, showing demonstrated improved performance and faster convergence compared to traditional optimizers. AdaGC stands out for its ability to adapt to gradient behavior during training, making it a promising alternative for applications requiring efficient and robust optimization.

Liu et al. (2024) developed an objective metric and benchmark dataset for quality assessment of multi-exposure fused light field images (MEFLFIs) in dynamic scenes. Their work introduces a new evaluation method that combines local-global joint features and angular quality-aware features to better align the objective metric with human visual perception. Experiments showed that the proposed metric significantly outperforms other state-of-the-art metrics for light field images, achieving high correlation with subjective assessments. The main contribution lies in providing a robust benchmark and an adaptive metric for quality evaluation in complex image fusion scenarios.

A recent contribution by Jamshidi et al. (2025) proposes SecuEdge-DRL, a self-adaptive cyber defense model for sustainable IoT environments. This approach integrates deep reinforcement learning (DRL) with the MAPE-K control loop to enable real-time intrusion detection and adaptive response without relying on predefined data models. SecuEdge-DRL focuses on optimizing both security and energy efficiency, addressing critical challenges in IoT edge computing. The model implements targeted security policies against various cyber threats such as DoS and DDoS attacks, achieving an average detection accuracy of 92% across diverse real-world scenarios. This work complements Oraculum by emphasizing security-driven self-adaptation in resource-constrained IoT systems, highlighting the growing importance of integrating cybersecurity and sustainability in adaptive architectures.

None of the surveyed approaches explicitly explored the use of deterministic policy gradient methods such as the *Twin Delayed Deep Deterministic Policy Gradient (TD3)* in self-adaptive system control for smart environments. Existing reinforcement learning-based models, including those built on Soft *Actor-Critic (SAC)* or *Proximal Policy Optimization (PPO)*, generally rely on stochastic policy updates and shared critic networks, which can introduce overestimation bias and slower convergence in dynamic IoT conditions. Oraculum addresses these limitations through three improvements introduced by TD3: twin critic networks that mitigate overestimation errors, delayed policy updates that enhance training stability, and target policy smoothing that prevents noise exploitation during learning. Combined with the ontology-backed metric space described above, these mechanisms enable Oraculum to anticipate context changes across heterogeneous domains and to achieve lower MAT and higher stability than architectures such as AS-MAC (Sah et al., 2022) and CLA-PSO (Velrajan & Sharmila, 2023) under the evaluation scenarios considered in this work.

The Oraculum model offers a unified approach to manage adaptation in smart environments through its combination of learning-based techniques. Rather than relying on static adaptation policies or domain-specific rules, it employs a dynamic decision process that continuously balances accuracy, efficiency, and stability. This approach allows it to generalize across heterogeneous IoT contexts, making it suitable for operational scenarios where adaptability, modularity, and low latency are essential.

3. Oraculum model

The Oraculum defines a modular architecture for self-adaptive systems operating in dynamic and heterogeneous environments. Its design integrates monitoring, prediction, decision-making, and execution mechanisms within a unified MAPE-K-based model. The model structure is organized into distinct functional modules, each responsible for a specific adaptation task. The workflow includes continuous system monitoring, identification of behavioral patterns through predictive analytics, and policy-driven adaptations executed via a RL agent. The architecture supports interaction among modules through asynchronous data flows and decision triggers. Subsequent subsections present the model's architecture and workflow, outline supported adaptation scenarios, and describe the roles of individual components.

The Oraculum model is designed to operate in dynamic and heterogeneous smart environments, particularly those involving IoT devices and distributed systems. The system consists of multiple interconnected components such as IoT devices, edge nodes, and cloud services, which generate and exchange performance metrics related to hardware (CPU, memory, GPU usage), network (latency, packet loss, throughput), software (response time, garbage collection), energy consumption, and SLA compliance. The environment is assumed to be partially observable and stochastic, with metrics collected at regular intervals. Data losses and transmission delays may occur but are mitigated through heuristic data aggregation and filtering. The system operates under constraints such as latency bounds, energy budgets, and SLA targets, which guide the reward formulation and adaptation policies.

While the Oraculum model conceptually considers potential threats including operational anomalies and adversarial conditions such as denial-of-service attacks and intrusion attempts, the current prototype does not encompass security-related metrics and therefore does not analyze attack scenarios. However, the model integrates semantic reasoning and reinforcement learning mechanisms that can be extended to detect and mitigate such threats proactively. This formalization provides a comprehensive understanding of the operational context and informs the design of predictive models, semantic ontology integration, and adaptive decision-making within the Oraculum.

3.1. Model overview

The Oraculum defines an adaptive architecture that integrates data processing, predictive modeling, and automated decision-making for intelligent IoT-based environments. The model incorporates regression, classification, and RL techniques to support runtime adaptation. Fig. 1 presents the complete workflow, including the modules and their interactions.

The adaptation process is distributed across the following modules:

1. **API Collector:** Collects performance metrics from IoT components to enable continuous observation of system behavior.
2. **Save Metrics:** Stores collected metrics in structured datasets for future retrieval and longitudinal analysis.
3. **Metric Ontology Integration:** Applies semantic reasoning using ontology-based models to interpret metrics and detect operational anomalies.
4. **Regression AI:** Predicts future metric values using historical data patterns to support proactive decision-making.
5. **Classification AI:** Categorizes predicted values to identify deviations from expected behavior.
6. **Possible Future High Value in Some Metric:** Identifies predicted values that exceed defined thresholds, indicating the need for intervention.
7. **Generate an Alert:** Issues alerts when forecasted values surpass acceptable limits.
8. **Create a Trigger:** Converts alerts into executable adaptation instructions.
9. **Update RL Engine:** Updates the RL model using new observations to improve policy accuracy.
10. **RL Agent Chooses an Action:** Selects adaptation actions based on current conditions and the learned policy.
11. **Check if Action is Still Required:** Revalidates the selected action against updated predictions and classifications to avoid redundant execution.
12. **Execute the Actions:** Applies verified adaptation strategies within the operational environment.
13. **Save the Action:** Records executed actions to maintain a log of adaptation history.
14. **Dashboard:** Displays system metrics, active alerts, adaptation status, and performance indicators for monitoring and administrative access.

The architecture integrates forecasting and decision modules to anticipate performance deviations. Predictions provide early input to the RL agent, which selects candidate actions based on historical context and observed behavior. Subsequent validation ensures that only relevant actions are executed, reducing unnecessary adaptations (Zeshan et al., 2023). Updates to the RL engine refine future decisions, while the dashboard module presents operational feedback in real time. This process supports anticipatory adaptation and consistent oversight in different run-time scenarios.

3.2. Model architecture

The Oraculum adopts the Technical Architecture Modeling (TAM) methodology, as proposed by SAP (2007). TAM extends the Unified Modeling Language (UML) to describe software architectures through structured diagrams and standardized elements. Fig. 2 presents the Oraculum architecture using the TAM model, indicating the responsibilities and interactions among components.

The architecture consists of interconnected modules that support adaptive behavior through metric acquisition, semantic processing, predictive modeling, decision-making, and action execution (Jaskierny & Kotulski, 2023).

The *API Collector* module acts as the initial point of integration with monitored IoT components. It retrieves performance metrics from distributed sources and channels them through a sequence of subcomponents. The Request Service coordinates incoming data requests, the Data Treat module transforms raw inputs into structured values, the Metric Map standardizes key-value pairs into recognized metrics, and the Record Builder constructs consistent data records. These records are then stored in two forms: Current Data, reflecting the most recent values for real-time use, and Data Persistence, which archives complete histories for analytical access.

The *Database* component classifies and stores incoming records in distinct layers: Raw Data for unprocessed values, Time Series for chronological data sequences, and Aggregated Metrics for preprocessed statistical summaries. This structure enables efficient access for model training and comparison between historical and real-time patterns, forming the

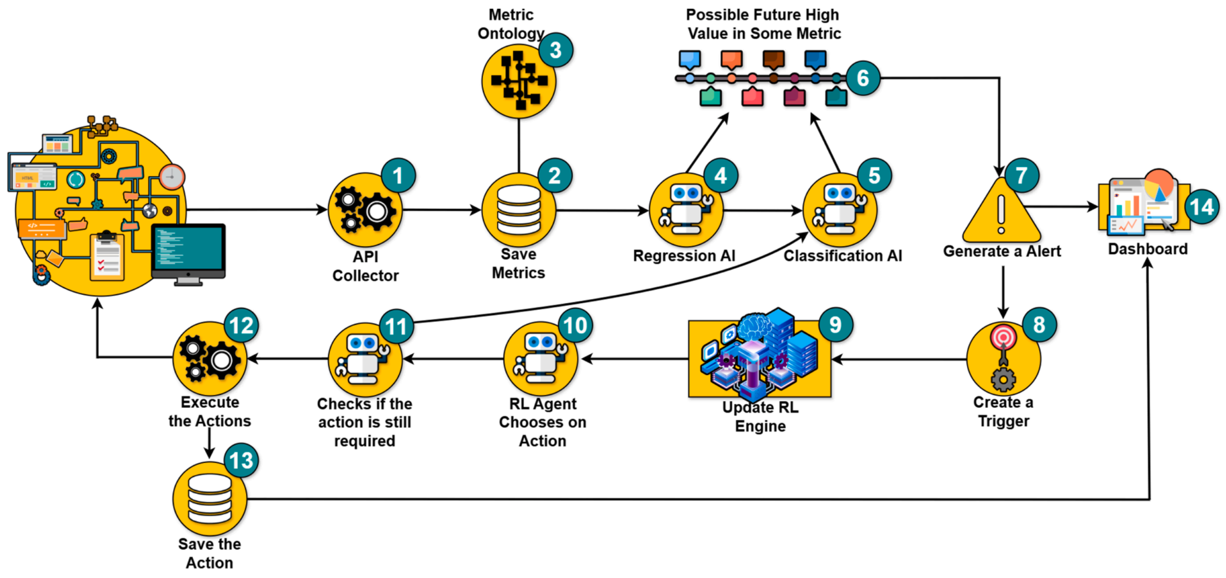

Fig. 1. Overview of the Oraculum model.

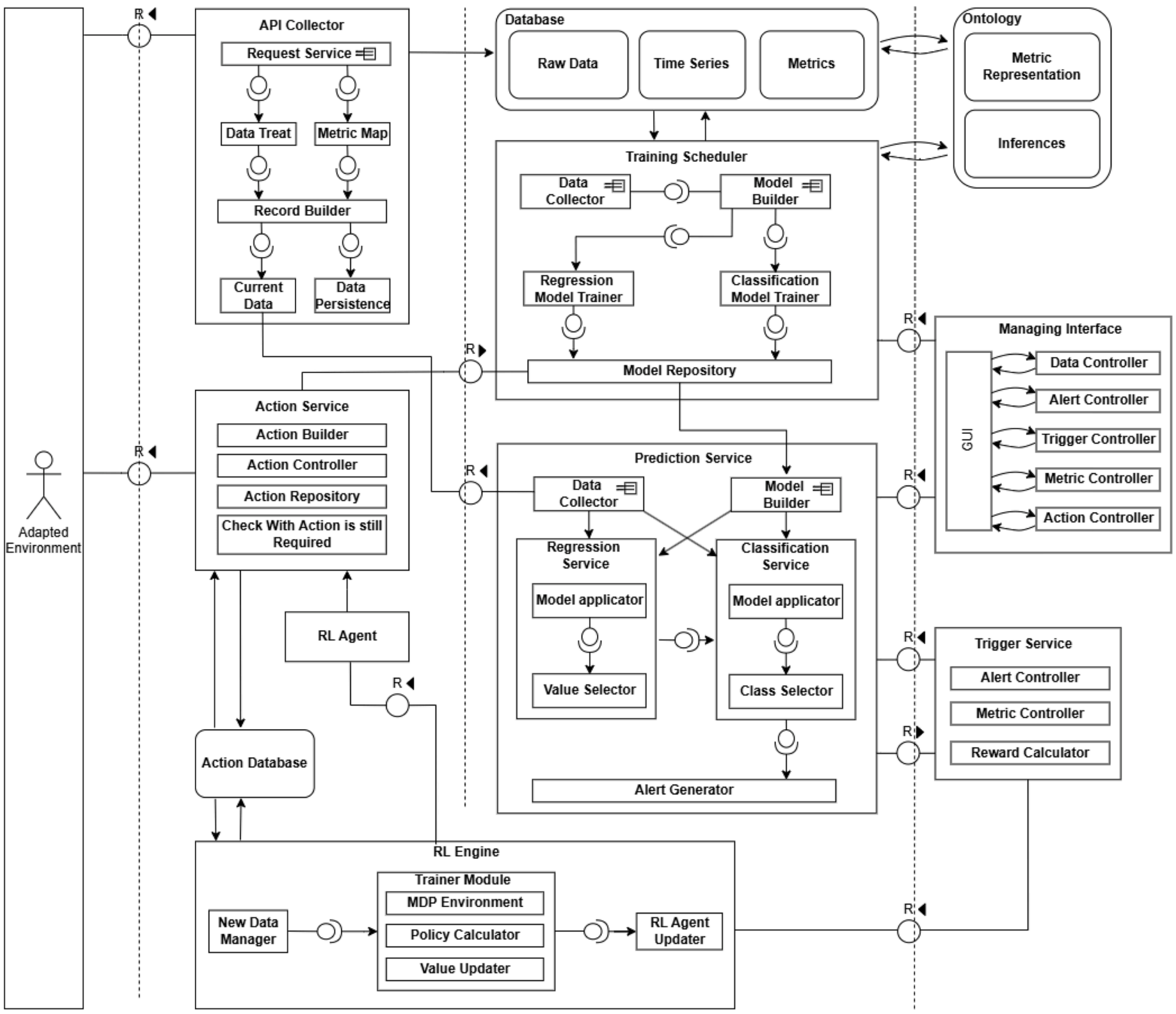

Fig. 2. Oraculum model architecture.

data backbone for adaptive logic. The *Ontology* module applies semantic reasoning to the collected metrics. Its Metric Representation defines hierarchical, causal, and associative relationships among monitored parameters. The Inference Engine processes these relations through logical rules to determine the system's operational state and classify events as normal or anomalous. This knowledge is shared with the RL Engine to guide the learning and evaluation of actions, ensuring that semantic dependencies influence the decision-making process.

The *Training Scheduler* governs the construction and updating of predictive models. It periodically extracts data from the database, formats input sets according to model requirements, and applies machine learning algorithms to produce regression and classification models. Trained models are stored in the Model Repository and updated regularly to reflect evolving workload characteristics. The *Prediction Service* applies these trained models to incoming data streams, estimating future metric values and labeling predicted conditions as normal or anomalous. When classifications exceed defined thresholds, alerts are generated and passed to the adaptation layer.

The *Trigger Service* manages the transformation of alerts into actionable instructions. The Alert Controller validates each alert, while the Metric Controller identifies affected parameters. The *Reward Calculator*, in coordination with the Ontology module, assigns utility values to previous actions based on semantic context and historical outcomes. This interaction ensures that knowledge-driven reasoning complements reinforcement-based policy learning.

The *RL Engine* implements the reinforcement learning process for adaptive policy generation. It is composed of three main components: the Trainer Module simulates state-action transitions in a *Markov Decision Process (MDP)* environment; the *Policy Calculator* identifies optimal decisions based on expected long-term rewards; and the *Value Updater* refines learned policies using observed outcomes. The RL Engine continuously receives context annotations from the Ontology module (e.g., metric relevance and causal relationships) that help prioritize actions with greater semantic impact. In return, the ontology receives the results of executed adaptations to update its knowledge base, enabling bidirectional feedback between symbolic reasoning and policy-based learning.

The *RL Agent* uses the policy generated by the RL Engine and the contextual semantics provided by the Ontology to select the most appropriate adaptation action. The chosen action is forwarded to the *Action Service*, which prepares, validates, and executes the operation through its submodules (Action Builder, Controller, and Repository). Before final execution, the action is revalidated to avoid redundant or contradictory operations, ensuring decision consistency.

The *Action Database* records all executed actions and their resulting outcomes. This log supports retrospective evaluation and further refinement of the RL policy through reward feedback loops. The *Managing Interface* provides an interactive visualization layer for administrators to monitor collected metrics, analyze alerts, and observe active adaptations in real time, reinforcing transparency and enabling supervisory control.

Fig. 3 illustrates the Multi-Agent System, elaborated using the Prometheus methodology (Padgham & Winikoff, 2004). This methodology defines a systematic model for the design of intelligent agent systems, emphasizing modular organization and explicit interactions among components. The diagram outlines the processes and interrelations that compose the model's adaptation cycle.

The Oraculum adopts an agent-oriented structure in which functionalities are encapsulated into modular and interdependent units. The process begins with the *API Collector*, which acquires performance metrics from monitored system elements. The data undergoes preprocessing operations-including transformation, mapping, and record structuring-and is then persisted in a *Time Series* database. In parallel, the *Ontology* module processes collected data to infer semantic relationships, perform knowledge-based queries, and enrich the dataset with contextual attributes.

The predictive analytics pipeline incorporates dynamic model training and inference. The *Training Scheduler* manages the continuous update of regression and classification models using new datasets. These models are stored and applied by the *Regression Predict Service* and *Classification Predict Service*, which respectively estimate future metric trends and classify operational states. When predicted values surpass specified thresholds, the system generates alerts that trigger subsequent adaptation phases.

The *Trigger Service* evaluates alert conditions and determines whether intervention is required. It incorporates RL mechanisms to associate previous decisions with outcome-based reward values. When adaptation is warranted, the service initiates a structured response by passing the context to the *RL Engine*. This component refines adaptation policies through iterative learning in a Markov Decision Process (MDP) environment. The *RL Agent* selects an appropriate action based on the current system state and learned policy, forwarding the decision for execution.

The architecture incorporates feedback mechanisms for post-adaptation evaluation. Each executed action is logged in the *Action Historic* database to support future analysis and policy adjustments. The *Managing Interface* provides visualization and control mechanisms, enabling observation of system status, review of adaptation history, and modification of operational parameters.

Fig. 4 illustrates the ontology developed to support structured classification of performance metrics. The model defines six primary categories used to interpret and organize system characteristics in the Oraculum architecture (Emmanuel Sapnken & Gaston Tamba, 2022; Shi et al., 2025). The ontology module plays an important role in the adaptation process by generating the initial labeled datasets used to train the classification models. By semantically categorizing raw performance data into meaningful classes, the ontology provides a structured representation of metric behavior and context, which is then consumed by the prediction and decision-making modules as part of Oraculum's end-to-end workflow.

The ontology classifies monitored metrics into the following categories: *Software*, *Hardware*, *Network*, *Energy*, *Security and Reliability*, and *Service Level Agreements (SLA)*. The *Software* category includes metrics such as response time, garbage collection duration, and throughput, which relate to application behavior and processing efficiency. *Hardware* encompasses CPU, memory, and GPU utilization metrics that support the assessment of physical resource usage. *Network* metrics address conditions such as packet loss, jitter, and transfer rates, which reflect communication quality between system components. *Energy* comprises indicators like power consumption and battery status, relevant for evaluating resource-constrained operations. *Security and Reliability* category includes detection rates, error rates, and log event monitoring, used to identify anomalies and faults in runtime execution. *SLA* metrics–such as availability, resilience, and stability–support compliance assessment relative to defined quality-of-service targets.

It is important to highlight that the ontology is not an auxiliary feature but a structural component of Oraculum's architecture. From the initial design, all monitored performance metrics were processed and classified through the ontology to ensure relational consistency among heterogeneous data sources. This semantic layer enables the model to establish dependencies between metrics that belong to different categories (e.g., linking CPU usage with response latency or packet loss with availability), which would be difficult to capture using isolated statistical or heuristic methods. As a result, the ontology constitutes the foundation that allows Oraculum to reason about metric relationships and execute context-aware adaptive decisions. The system, therefore, was developed and validated with the ontology fully embedded in the pipeline, and no comparative baseline without the ontology was implemented.

Within Oraculum, the ontology provides a formal mechanism for metric classification and interpretation. The Ontology Module maps raw performance data into structured categories and applies rule-based inference to identify correlations and dependencies. This classification

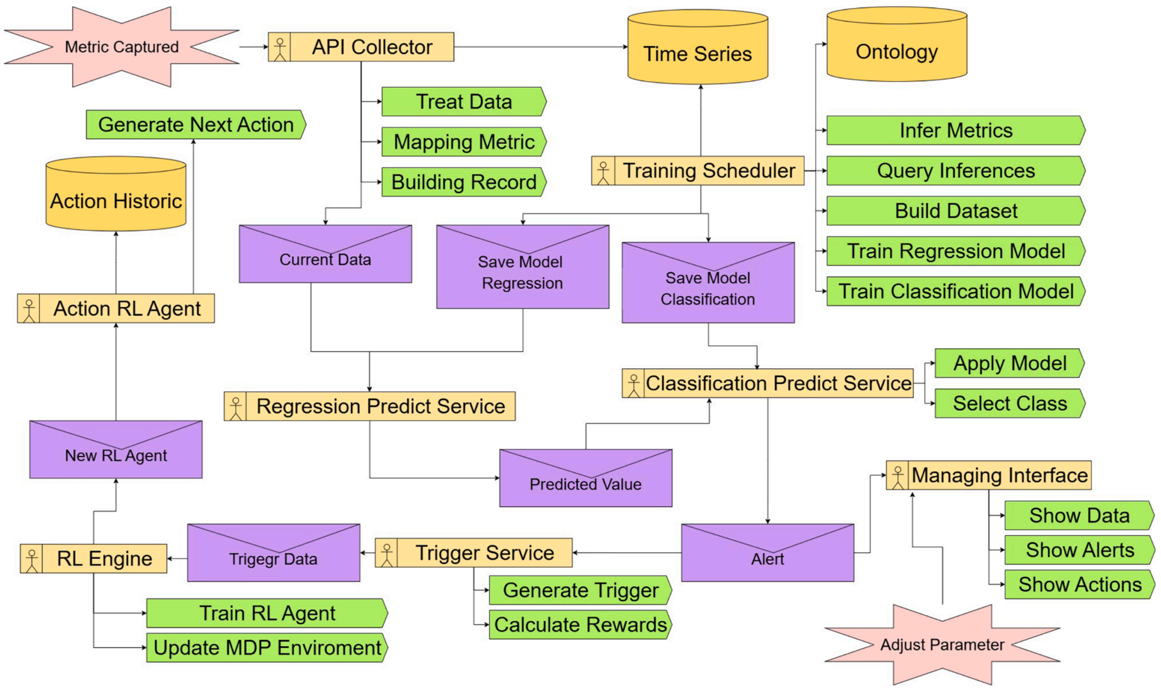

Fig. 3. Oraculum model following prometheus methodology.

Table 2
Configurable parameters in the Oraculum model.

Parameter	*Description*	*Data Type*	*Possible Values*	*Default Value*
Monitored Metrics	Defines which system metrics are observed	Set of Strings	User-defined	CPU, Memory, Latency
Metric Collection Interval (T_c)	Time between metric samples	Integer (seconds)	≥ 1	10 sec
Prediction Horizon (T_p)	Time window for forecasting	Integer (seconds)	≥ 1	60 sec
Alert Sensitivity (θ)	Threshold for triggering alerts	Float [0,1]	$\{0.25, 0.5, 1.0\}$	0.5
RL Action Aggressiveness (A_t)	Degree of action intensity over time	Function	$\{A_0, e^{-\lambda t}, \min\}$	$A_0 e^{-0.1t}$
RL Policy Update Algorithm	Learning algorithm for adaptation	String	TD3, SAC, PPO, Q-learning	TD3
RL Actions	Set of actions taken by the model	List of Actions	Scaling, Scheduling, Optimization	Scaling (CPU, Memory)

supports context-aware prediction by enhancing the semantic representation of metric behavior. The structured ontology facilitates metric-based adaptation by informing both anomaly detection and RL-based decision-making. The model uses this classification to guide alert generation and policy selection during adaptive processes, but the present evaluation does not include an ablation scenario in which the ontology is removed or replaced.

3.3. Model parameters

The Oraculum includes configurable parameters that define how the system collects, analyzes, and reacts to performance metrics. These parameters influence aspects such as sampling frequency, forecasting range, alert sensitivity, and adaptation aggressiveness. Table 2 summarizes the parameters used in the current implementation, including their data types, valid values, and defaults.

The complete set of monitored performance metrics is denoted as $M = \{m_1, m_2, \ldots, m_n\}$, specifies the performance indicators analyzed by the system. Each metric m_i belongs to a category such as resource utilization, latency, security, or energy efficiency. The selection of metrics directly influences adaptation decisions and the scope of operational awareness.

The system samples metrics at intervals defined by the collection parameter T_c, balancing data granularity with processing overhead. Forecasting operations rely on a prediction horizon T_p, which defines the number of future time steps estimated using historical input $X = \{x_1, x_2, \ldots, x_t\}$. The regression function f produces estimates according to:

$$\hat{x}_{t+k} = f(X, k), \tag{1}$$

where k is the forecasting window. Shorter horizons provide near-term estimates, while longer horizons enable broader trend prediction with reduced accuracy. Alert generation depends on classifier agreement across N models and a threshold $\theta \in [0, 1]$. The system determines alert activation using the rule (as illustrated in the Classification AI module of Fig. 1):

$$A = \begin{cases} 1, & \text{if } \sum_{i=1}^{N} C_i(x) \geq \theta N, \\ 0, & \text{otherwise}, \end{cases} \tag{2}$$

where $C_i(x)$ is the classification outcome of model i. A lower θ increases alert frequency, while a higher value restricts activation to events with greater classifier consensus.

RL behavior is modulated using a configurable aggressiveness function. The system reduces the intensity of adaptations over time based on an exponential decay model:

$$A_t = A_0 e^{-\lambda t}, \tag{3}$$

where A_t is the adaptation magnitude at time t, A_0 is the initial value, and λ is the decay rate. The exponential decay function $A_t = A_0 e^{-\lambda t}$ was chosen due to its widespread use in modeling diminishing effects over time in adaptive control systems. The parameter λ controls the

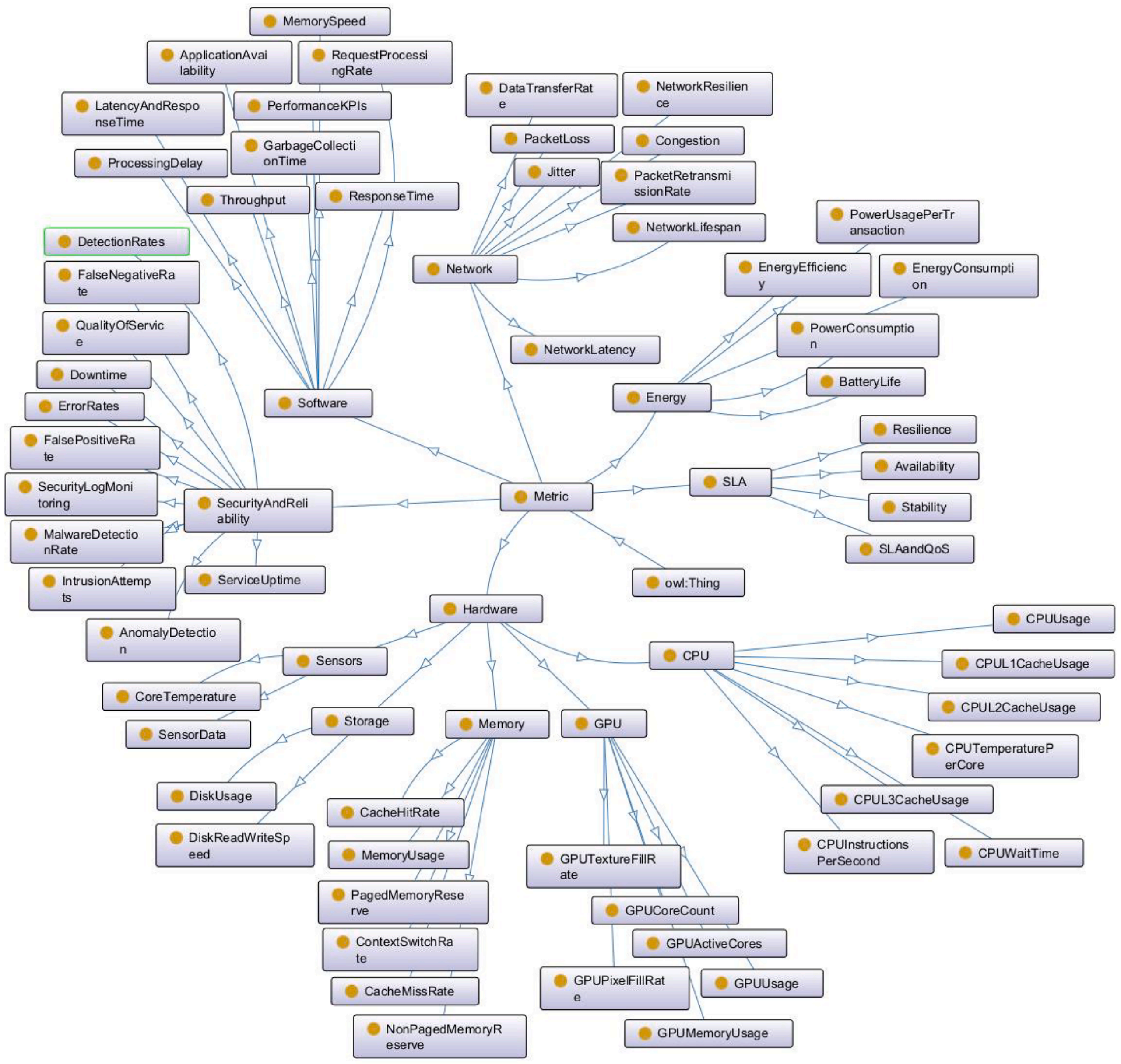

Fig. 4. Ontology representation for performance metrics.

rate of decay and was empirically tuned through preliminary experiments. Values of λ were explored over a range to balance adaptation speed and stability, and the value that maximized system responsiveness without inducing oscillations or premature convergence was selected.

The model supports multiple reinforcement learning (RL) algorithms for policy updates, including TD3, SAC, PPO, and Q-learning. Each algorithm exhibits distinct behaviors regarding convergence rate, sample efficiency, and stability during training. After preliminary experiments using the SHIELD environment, TD3 was selected as the primary policy update strategy due to its stability and faster convergence in continuous control domains. Unlike Q-learning, which is limited to discrete actions, or PPO and SAC, which showed higher gradient noise and slower convergence, TD3 effectively mitigates overestimation bias and ensures smoother policy evolution through the use of double critics and delayed policy updates. These properties make TD3 better suited for the fine-grained continuous adaptation cycles required in Oraculum. The adaptation process executes actions such as scaling, scheduling, and optimization, which can be parameterized to align with deployment constraints or operational priorities.

The ability to configure these parameters enhances the flexibility of the Oraculum across different domains. The combination of predictive modeling, RL-based adaptation, and customizable parameters ensures that the system maintains optimal performance under dynamic workloads.

4. Implementation aspects

The Oraculum prototype was implemented to evaluate adaptation performance under controlled conditions. The implementation process included the configuration of operational parameters, specification of monitoring strategies, and deployment of adaptive mechanisms in a testbed environment. The SHiELD simulator provided a dynamic execution context for validating self-adaptive behavior. The prototype continuously monitored performance metrics and modified system

Table 3
Configurable actions for RL agent.

Action Type	*Description*
Horizontal Scaling	$n_{new} = n_{current} + \Delta n$, where Δn is the number of added/removed nodes.
Vertical Scaling	$\text{Resource}_{new} = \text{Resource}_{current} + \Delta R$, modifying CPU, memory, or disk.
Adaptive Scheduling	Adjusts scheduling priorities dynamically based on workload demand.
Data Processing Optimization	Changes data filtering, aggregation, or compression parameters dynamically.
Custom Linux Commands	Executes predefined shell scripts for system-specific actions.

configurations based on predicted anomalies to improve resource allocation and runtime stability.

The monitoring phase included metrics across four dimensions: hardware, network, software, and service-level agreement (SLA). These indicators were selected to represent distinct aspects of system behavior relevant to adaptive operation:

- **Hardware:** CPU Usage, Memory Usage, GPU Usage, Storage Utilization
- **Network:** Latency, Data Transfer Rate, Packet Loss
- **Software:** Response Latency, Throughput, Garbage Collection Time
- **SLA:** Availability, Resilience, Stability

All metrics were normalized within the range [0.0, 1.0] to standardize input values and simplify processing during model execution. The Oraculum applied regression techniques to forecast metric trends and classification models to detect behavioral deviations. Predictive outputs informed preemptive adaptations by enabling system reconfiguration prior to observable degradation. A RL agent adjusted policy parameters over time using feedback from the environment, contributing to refined action selection and improved response efficiency.

The implementation is detailed in the following subsections. SHiELD Simulator describes the validation platform. Architecture Implementation presents the software architecture and module configuration. Data Collection outlines the metric acquisition and preprocessing steps. Regression and Classification Models explains the forecasting and anomaly detection processes. RL Agent details the mechanisms for self-adaptive action selection.

4.1. SHiELD simulator

The SHiELD (Sensor Heuristics and Intelligent Evaluation for Large-scale Data) simulator offers a flexible environment for evaluating self-adaptive strategies in IoT systems. It supports continuous sensor data generation, heuristic optimizations, predictive modeling, and fault injection, enabling comprehensive experimentation with monitoring, anomaly detection, and adaptive control.

Recent updates to SHiELD include the integration of multiple time series forecasting models (such as ARIMA, LSTM, GRU, and Transformer), automated fault injection mechanisms (e.g., latency, packet loss, disconnection), and the generation of detailed reliability reports. The simulator now also supports the evaluation of elasticity algorithms, such as the Kubernetes Horizontal Pod Autoscaler, by simulating dynamic resource allocation scenarios.

Heuristic techniques, like data aggregation, compression, and filtering, are applied to optimize transmission and reduce redundant information, minimizing bandwidth and computational overhead. Predictive models estimate future sensor values, supporting proactive adaptation and resilience strategies. Fault injection and recovery analysis allow for the assessment of system robustness under adverse conditions.

SHiELD consolidates all stages of sensor data management, from generation and transformation to forecasting and performance monitoring, within a unified architecture. This integrated approach enables consistent evaluation of adaptive mechanisms and system reliability across the entire sensing and response lifecycle, closely mirroring real-world IoT deployments.[1]

4.2. Architecture implementation

The Oraculum prototype operates within a containerized environment using Docker, with orchestration managed by Docker Compose. This architecture supports modular deployment, streamlines component integration, and provides isolation for monitoring and evaluation. Each service executes independently, enabling explicit control over resource allocation and inter-service communication. The architecture also incorporates mechanisms for capturing internal system metrics to assess the overhead associated with adaptive operations.

The monitoring subsystem collects runtime information from all components, including execution times, CPU and memory consumption, disk usage, and network transfer volumes. These measurements quantify the computational cost introduced by each adaptive mechanism and support comparative performance analysis (Caporuscio et al., 2016; Ding et al., 2025).

The prototype is composed of specialized nodes, each responsible for a distinct function in the adaptation pipeline. Table 4 describes the role and technologies associated with each component.

This architectural configuration provides a basis for assessing the performance and coordination of adaptive mechanisms. Internal monitoring tracks the behavior of each module, supporting the alignment between adaptation decisions and the system's operational requirements and constraints.[2]

4.3. Regression and classification models

The Oraculum employs regression and classification algorithms to process metric data collected during system operation. This process corresponds to steps 4 and 5 illustrated in Fig. 1. Datasets used for training are produced by the SHiELD simulator, which generates time-series observations across varying environmental configurations. Each dataset corresponds to a monitored metric, and multiple algorithms are evaluated to determine predictive accuracy and generalization capacity.

The model training pipeline comprises dataset generation, preprocessing, model selection, training, evaluation, and storage. For a dataset D with n samples and m features, the structure is defined as:

$$D = \{(X_1, y_1), (X_2, y_2), \dots, (X_n, y_n)\} \tag{4}$$

where $X_i \in \mathbb{R}^m$ denotes the feature vector and y_i is the target variable. The dataset is partitioned into training and testing subsets using an 80/20 split:

$$D_{\text{train}}, D_{\text{test}} = \text{split}(D, 0.8) \tag{5}$$

Two predictive tasks are defined in the learning process:

- **Regression:** Estimates future values of a monitored metric based on historical data. The regression function $f : \mathbb{R}^m \to \mathbb{R}$ approximates

[1] The complete source code and implementation details are available at: https://github.com/DarlanNoetzold/SensorSimulator.

[2] The complete source code and implementation details are available at: https://github.com/DarlanNoetzold/Oraculum-Model

Table 4
Oraculum model architecture components.

Component	Function	Technologies Used
API-Collector	Acquires system metrics in real time and stores them in a time-series database.	Spring Boot
Database	Maintains structured time-series data from collected metrics.	PostgreSQL
Ontology-Integration	Executes SWRL rules over OWL-based ontologies to identify anomalous conditions.	C, OWL, SWRL
Dataset-Builder	Constructs raw and labeled datasets from historical metrics for model training.	C
Training-Scheduler	Trains regression and classification models, selecting Trains regression and classification models, selecting the most effective ones.	Python, TensorFlow
Data-Collector	Retrieves the latest metrics for real-time inference.	Python
Prediction-Service	Applies trained models to current metrics for forecasting and anomaly detection.	Python, Flask
Trigger-Service	Forwards anomaly alerts to the RL module.	Spring Boot
RL-Engine	Builds and refines RL policies based on feedback.	Python
RL-Agent	Executes adaptation policies generated by the RL-Engine.	Python, Flask, Pre-trained RL Model
Action-Service	Validates and enforces adaptive actions according to system goals.	Spring Boot

the mapping $y = f(X)$. Algorithms explored include linear regression, decision trees, and neural networks.

- **Classification:** Identifies system states by labeling predicted values according to behavioral patterns. The classifier $g : \mathbb{R}^m \to \{0, 1\}$ determines whether observed conditions correspond to expected or anomalous states. Models tested include logistic regression, decision trees, and feedforward neural networks.

After training, models are evaluated using accuracy, precision, recall, F1-score, and area under the ROC curve (AUC), depending on the task. These metrics quantify the model's ability to make reliable predictions under varying operational scenarios. Regression models are additionally evaluated using root mean square error (RMSE) and mean absolute error (MAE), which measure the deviation between predicted and actual values over time.

The training and evaluation process follows a systematic pipeline as outlined in Algorithm 1. For each dataset corresponding to a specific node and metric, multiple model types, including linear regression, decision trees, and various neural network architectures, are trained and tested. Neural networks are trained using the Adam optimizer over a fixed number of epochs with mini-batch gradient descent, while traditional models are trained using standard fitting procedures. Performance metrics appropriate to the task type are computed and logged, enabling comprehensive comparison across models. Visualizations of model performance are generated to facilitate analysis and interpretation, supporting informed selection of the best-performing models for deployment.

Once validated, the models are integrated into the prediction pipeline. During runtime, regression models continuously forecast the evolution of monitored metrics within a predefined horizon, enabling the system to anticipate resource saturation, latency spikes, or performance degradation. Classification models process the forecasted values to determine whether projected behavior deviates from expected patterns, triggering alerts when thresholds are exceeded.

The outputs of the classification layer serve as input to the trigger module, which coordinates adaptation actions through the RL agent. This interaction ensures that predictive inferences contribute directly to the decision-making process, allowing the system to react before disruptions escalate. The prediction service is executed at fixed intervals, aligned with the metric collection cycle, and incorporates new data in a sliding window to maintain up-to-date context.

Each model is trained and stored independently for every monitored metric, supporting scalability. The modular design simplifies model updates and enables parallel inference when processing multiple metrics simultaneously. Periodic retraining with new operational data reassesses model performance, ensuring alignment with evolving workload patterns. This approach maintains consistency in anomaly detection and adaptation decisions. Data preprocessing improves model performance and consistency across different metrics:

Algorithm 1 Model training pipeline.

```
Require: Set of datasets $\mathcal{D} = \{D_1, D_2, \dots, D_n\}$, node name $N$, metric name $M$
1:  for each dataset $D \in \mathcal{D}$ do
2:      Load dataset $D$
3:      Split dataset into training ($X_{\text{train}}, y_{\text{train}}$) and testing ($X_{\text{test}}, y_{\text{test}}$) sets
4:      for each model type $T \in$ {Linear Regression, Decision Tree, NN1, NN2, NN3} do
5:          Initialize model based on $T$
6:          if $T \in$ {NN1, NN2, NN3} then
7:              Compile neural network model with Adam optimizer and appropriate loss function
8:              Train model on $X_{\text{train}}, y_{\text{train}}$ for 50 epochs, batch size 32
9:              Generate predictions on $X_{\text{test}}$
10:         else
11:             Train model on $X_{\text{train}}, y_{\text{train}}$
12:             Generate predictions on $X_{\text{test}}$
13:         end if
14:         Compute performance metrics (MAE for regression, Accuracy/AUC/F1 for classification)
15:         Log results including node $N$, metric $M$, and performance scores
16:         Save trained model
17:     end for
18:     Generate and save model performance visualization
19: end for
```

1. Handling Missing Data: Missing values are replaced using interpolation or mean imputation to prevent inconsistencies.
2. Feature Scaling: Min-max normalization standardizes all features to the range $[0, 1]$:

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{6}$$

3. Dataset Splitting: The dataset is divided into training and test sets using an 80-20% split.

Table 5
Selected models and parameter configurations.

Model	Parameter	Value
Linear Regression	Solver	Normal Equation
	Regularization	None
Decision Tree Regressor	Max Depth	10
	Min Samples Split	2
	Min Samples Leaf	1
NN1 - Regression	Layers	1
	Neurons per Layer	64
	Activation	ReLU
	Optimizer	Adam
	Loss Function	MSE
NN2 - Regression	Layers	2
	Neurons per Layer	128, 64
	Activation	ReLU
	Optimizer	Adam
	Loss Function	MSE
NN3 - Regression	Layers	2
	Neurons per Layer	64, 32
	Activation	ReLU
	Dropout	0.5
	Optimizer	Adam
	Loss Function	MSE
Decision Tree Classifier	Max Depth	10
	Min Samples Split	2
	Min Samples Leaf	1
Logistic Regression	Solver	LBFGS
	Regularization	L2
	Regularization Strength	1.0
NN1 - Classification	Layers	1
	Neurons per Layer	64
	Activation	ReLU
	Optimizer	Adam
	Loss Function	BCE
NN2 - Classification	Layers	2
	Neurons per Layer	128, 64
	Activation	ReLU
	Optimizer	Adam
	Loss Function	BCE
NN3 - Classification	Layers	2
	Neurons per Layer	64, 32
	Activation	ReLU
	Dropout	0.5
	Optimizer	Adam
	Loss Function	BCE

4. Data Balancing (for classification): Oversampling the minority class or undersampling the majority class improves classification performance when class imbalance is present.
5. Feature Selection: Correlation analysis and importance scores from decision trees help remove redundant features.

Table 5 summarizes the models and parameters adjusted during training. The models were selected to balance computational efficiency and predictive accuracy. Linear regression provides a baseline for detecting linear trends, while decision trees capture non-linear dependencies with controlled complexity. Neural networks are structured with increasing depth across different configurations (NN1 and NN2) to explore hierarchical feature extraction. Each trained model is stored and integrated into the Oraculum model, enabling real-time monitoring and predictive adaptation.

4.4. RL agent

RL is a machine learning paradigm in which an agent selects sequential actions in response to environmental conditions, receiving feedback in the form of rewards (Barrett et al., 2013). The interaction is modeled as a Markov Decision Process (MDP), where each state transition depends on the current state and selected action (Arias del Campo et al., 2021; Restuccia & Melodia, 2020).

In Oraculum, the RL agent operates within a Markov Decision Process (MDP) framework, where each system state is defined by a set of performance metrics such as CPU and memory usage, network latency, packet loss, and response time. The agent selects actions, such as adjusting resource allocations or scheduling policies, that directly influence the next system state. The transition between states is modeled as $s_{t+1} = T(s_t, a_t) + \epsilon$, where T is the deterministic system response and ϵ captures environmental uncertainty and noise. This stochastic component is essential for robust adaptation in real-world, dynamic environments.

The agent's objective is to maximize a cumulative reward, defined as a weighted sum of normalized performance metrics, thus balancing competing objectives like efficiency and responsiveness. The optimal policy $\pi^*(s)$ is learned using RL algorithms (e.g., SAC, TD3), which map observed states to actions that maximize expected returns. This approach enables the system to autonomously adapt to changing conditions, optimizing performance over time without manual intervention. The environment is defined by a set of system states $s_t \in S$, where each state comprises performance indicators:

$$s_t = \{m_1, m_2, \dots, m_n\}, \tag{7}$$

with m_i representing metrics such as CPU usage, memory consumption, latency, packet loss, response time, and throughput. The transition to the next state is defined by:

$$s_{t+1} = T(s_t, a_t) + \epsilon, \tag{8}$$

where $T(s_t, a_t)$ represents the deterministic component of the state transition and encodes how a specific adaptation action modifies the underlying system configuration and workload routing. Given a state–action pair (s_t, a_t), $T(s_t, a_t)$ produces the nominal next state s'_{t+1} that reflects the direct effect of the chosen action on resources, services, and quality-of-service indicators. The term ϵ then adds stochastic variations caused by exogenous factors such as workload spikes, hardware noise, and network fluctuations. This formulation separates the controllable effect of each adaptation decision, captured by $T(s_t, a_t)$, from the uncontrollable dynamics represented by ϵ, and therefore provides a structured view of how actions propagate through the system.

The state representation used by the RL agent (Eq. (8)) consists of raw metric values collected from heterogeneous sources, including hardware, SLA, software, and network parameters. This design encodes the relevant information directly in the state vector without additional aggregation or feature engineering, which simplifies the implementation and preserves fine-grained details about system behavior. However, this choice may limit learning efficiency in high-dimensional or noisy environments, because the agent must infer useful abstractions implicitly during training. Future work will explore state abstraction techniques such as PCA, autoencoders, and attention mechanisms to compress redundant signals, highlight the most influential features, and improve convergence speed.

The RL agent's objective is to optimize overall system performance by balancing these heterogeneous metrics, which often have conflicting goals (e.g., minimizing latency while maximizing throughput). The reward function integrates multiple metrics through weighted summations of normalized values. This allows the agent to learn policies that effectively trade off competing objectives. Weights can be tuned to prioritize certain metrics depending on operational requirements.

Currently, the state representation consists of raw metric values, which simplifies implementation but may limit learning efficiency in high-dimensional and noisy environments. This approach does not incorporate feature extraction or dimensionality reduction techniques, which could help the agent focus on the most relevant information and improve convergence speed. Future work will investigate methods such as principal component analysis (PCA), autoencoders, or attention mechanisms to create more compact and informative state representations.

Moreover, the current reward design employs fixed weights for balancing the different performance metrics, which provides a clear and interpretable optimization objective aligned with the system's operational goals. While this approach has proven effective in the evaluated scenarios, future work could explore adaptive weighting schemes or multi-objective RL techniques that explicitly consider Pareto front exploration. Such enhancements may offer increased flexibility and robustness in dynamically changing environments, further improving the model's ability to balance competing objectives under varying conditions.

Regarding the stochastic component ϵ in the MDP transition model (Eq. (9)), it represents environmental uncertainty and measurement noise inherent in real-world systems. Although acknowledged, its statistical properties were not explicitly modeled in the current prototype. Empirical observations suggest that ϵ follows a bounded noise distribution with zero mean, but further characterization and incorporation into the RL training process are planned to enhance robustness against uncertainty. The reward function guides the optimization process:

$$R(s_t, a_t) = \sum_{i=1}^{n} w_i f_i(m_i), \tag{9}$$

where w_i defines the importance of each metric and $f_i(m_i)$ normalizes its contribution. Each metric m_i is normalized and transformed by a function $f_i(m_i)$ tailored to its characteristics. For latency-related metrics, f_i is defined as an inverse function to penalize higher values, e.g., $f_i(m_i) = \frac{1}{1+m_i}$. For throughput or performance metrics, f_i is proportional to the metric value, e.g., $f_i(m_i) = \frac{m_i}{m_{\max}}$, where $m_{\max}$ is the maximum observed value. This design ensures the reward function appropriately balances conflicting objectives across heterogeneous metrics. The agent aims to maximize the cumulative discounted reward:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}), \tag{10}$$

where $\gamma \in [0, 1]$ is the discount factor. The agent uses multi-objective optimization to manage trade-offs between metrics. When improvements in one metric degrade others, the agent approximates Pareto-optimal solutions:

$$\nexists s' \in S, \quad \text{such that} \quad R(s') > R(s) \quad \forall s \neq s'. \tag{11}$$

A weighted objective function guides the optimization process:

$$J(s) = \sum_{i=1}^{n} \lambda_i f_i(m_i), \tag{12}$$

where each m_i is a performance metric observed in the current state s, and $f_i(m_i)$ is a transformation function applied to the metric m_i to normalize or scale it appropriately for aggregation. For example, f_i may be an inverse function for latency metrics (to penalize higher values) or a proportional scaling for throughput metrics (to reward higher values). The weights λ_i represent the relative importance of each metric and can adapt over time based on system priorities or observed conditions. This formulation allows the agent to balance multiple, potentially conflicting objectives in a single scalar reward signal. A state s^* is considered stable when small changes in the system do not lead to improvements in the objective function $J(s)$, indicating convergence to an optimal or satisfactory operating point. A state s^* is considered stable when marginal changes do not yield improvements in the objective function:

$$\nabla_s J(s^*) = 0, \quad \text{where} \quad J(s) = \mathbb{E}[G_t | s_0 = s]. \tag{13}$$

The optimal policy is derived from the action-value function:

$$\pi^*(s) = \arg\max_{a\in A} Q^*(s, a), \tag{14}$$

with $Q^*(s, a)$ estimated through RL algorithms that support continuous control. The implementation compares algorithms based on the type of policy, the exploration strategy, and the response to the dynamics of the system (Tang et al., 2024). Table 6 summarizes these characteristics.

Table 6
Comparison of RL algorithms for adaptation in the Oraculum model.

Algorithm	Policy Type	Exploration Strategy	Adaptability Level
Q-learning	Discrete	ϵ-greedy	Low
PPO	Continuous	Clipped objective	High
SAC	Continuous	Entropy regularization	Higher
TD3	Continuous	Target smoothing	Higher

Rather than converging to a static policy, the RL agent continuously updates its decisions to accommodate changes in system state and external conditions. The iterative optimization process adapts the policy over time, supporting resilient system behavior under fluctuating workloads and performance constraints.

The balance between exploration and exploitation was regulated by the learning algorithm selected for each experiment. Algorithms such as SAC and TD3 include entropy regularization and target smoothing, respectively, which influence the diversity of actions selected during policy optimization. Excessive exploration may lead to unstable behavior during high-load phases, while overly exploitative policies can converge prematurely to suboptimal configurations. Incremental adjustment of entropy coefficients during training maintains exploratory behavior in early phases and promotes convergence in later stages. This strategy enhances policy robustness while minimizing unnecessary adaptations.

Table 7 details the key hyperparameters used for each RL algorithm in the experiments. These hyperparameters were selected empirically through preliminary tuning to balance learning stability, convergence speed, and adaptability to the Oraculum environment's dynamic conditions.

The learning rate controls the step size during gradient updates, with smaller values promoting stable convergence but slower learning. The discount factor balances the importance of immediate versus future rewards, set close to 1 to emphasize long-term performance. Batch size and replay buffer size influence the diversity and stability of training samples, with larger buffers helping to decorrelate experiences. Policy update frequency and target smoothing coefficients in SAC and TD3 help stabilize training by controlling how often and how smoothly the policy is updated. The entropy coefficient in SAC encourages exploration by adding randomness to the policy, which is auto-tuned during training to balance exploration and exploitation dynamically.

The neural network architectures for PPO, SAC, and TD3 consist of fully connected layers with ReLU activations, sized to provide sufficient capacity for modeling complex policies without overfitting. Q-learning uses a tabular approach suitable for discrete state-action spaces. These hyperparameters were chosen based on a combination of literature recommendations and empirical tuning on preliminary experiments within the Oraculum environment. This approach ensured that each algorithm could effectively learn adaptive policies under the system's dynamic and heterogeneous conditions.

The sensitivity study assessed how different hyperparameter settings affect the reinforcement learning agent's behavior. The analysis varied parameters such as learning rate, discount factor, batch size, and exploration noise within representative ranges around the tuned values shown in Table 7. The learned policies and reward trajectories changed minimally across configurations. TD3 kept stable convergence and consistent rewards, while SAC and PPO showed only minor oscillations under the same variations. These results demonstrate that the RL component in Oraculum maintains stable and coherent decision-making under moderate hyperparameter changes, confirming the robustness of the adaptive process.

4.4.1. State space (S)

The state space S defines the system's current operational context based on a set of performance metrics. These metrics characterize mul-

Table 7
Key hyperparameters for RL algorithms used in Oraculum.

Parameter	Q-learning	PPO	SAC	TD3
Learning rate	1×10^{-3}	3×10^{-4}	3×10^{-4}	3×10^{-4}
Discount factor (γ)	0.95	0.99	0.99	0.99
Batch size	N/A (tabular)	64	100	100
Replay buffer size	100,000	N/A	1,000,000	1,000,000
Policy update frequency	N/A	N/A	Every step	Every 2 steps
Target smoothing coefficient (τ)	N/A	N/A	0.005	0.005
Entropy coefficient	N/A	N/A	Auto-tuned	N/A
Network architecture	N/A	2 layers, 64 units	2 layers, 256 units	2 layers, 256 units
Activation function	N/A	ReLU	ReLU	ReLU
Exploration strategy	ϵ-greedy	Clipped surrogate	Entropy regularization	Target policy smoothing

tiple system dimensions, including hardware, network, software, and service-level performance. Each state $s_t \in S$ aggregates real-time values used by the RL agent to determine adaptation actions.

The monitored metrics are grouped into the following categories:

- **Hardware**: CPU usage, memory usage, GPU utilization, and storage utilization. These indicators represent computational resource availability and consumption.
- **Network**: Latency, data transfer rate, and packet loss. These values reflect communication throughput and reliability between distributed system components.
- **Software**: Response time, throughput, and garbage collection duration. These features capture internal processing characteristics and execution efficiency.
- **Service Level Agreements (SLA)**: Availability, resilience, and stability. These attributes relate to compliance with quality-of-service objectives defined by operational requirements.

Each metric is normalized to the interval $[0.0, 1.0]$, enabling standardized evaluation across heterogeneous sources. The RL module interprets this representation to track system conditions and apply configuration changes as necessary to maintain performance within predefined operational bounds.

4.4.2. *Possible actions in the RL agent*

In the defined Markov Decision Process (MDP), the action space A includes a set of discrete and parameterized operations that modify system configurations. Each action is designed to influence performance-related variables across hardware, network, software, and data management dimensions. The selection of actions is informed by the current system state and bounded by predefined operational constraints to preserve system stability. Table 8 summarizes the available actions and corresponding parameter ranges. The RL agent selects from the following action categories:

- **Scaling Operations**: Adjusts computational resources, including CPU cores, memory allocation, GPU usage, and storage limits, in response to workload variations.
- **Node Management**: Initiates node restarts when performance degradation indicators exceed acceptable thresholds, aiming to restore service responsiveness.
- **Heuristic Adjustments**: Modifies parameters for data filtering, aggregation, and compression. These adjustments reduce data processing overhead while preserving relevant information for downstream analytics.
- **Resource Optimization**: Increases or decreases CPU and memory allocation dynamically, based on predictive assessments of current and near-future system demands.

4.4.3. *Reward definition (R)*

The reward function quantifies the impact of each action taken by the RL agent based on system performance. The function ensures that bene-

Table 8
Available actions and limits in the docker RL agent.

Action	Effect	Limit
Scale up CPU	Increase vCPUs by 1	Max +4 vCPUs
Scale up Memory	Increase RAM by 512MB	Max +8GB
Scale up GPU Memory	Allocate additional 512MB GPU memory	Max +8192MB
Scale up Storage	Expand disk space by 10GB	Max +500GB
Scale down CPU	Decrease vCPUs by 1	Min 1 vCPU
Scale down Memory	Decrease RAM by 512MB	Min 512MB
Scale down GPU Memory	Deallocate 512MB GPU memory	Min 512MB
Scale down Storage	Reduce disk space by 10GB	Min 20GB
Restart Node	Reboot instance	No limit
Migrate to New Instance	Move workload to another VM	No limit
Adjust Filtering	Modify data filtering threshold	Range [10%, 50%]
Adjust Aggregation	Modify data block aggregation	Range [5, 50]
Adjust Compression	Modify data compression ratio	Range [0.2, 0.9]
Change Autoscaling Policy	Modify scaling strategy	Predefined policies
Optimize Load Balancing	Adjust traffic distribution	Dynamic adjustment

ficial actions receive positive rewards, while actions that degrade performance or unnecessarily increase resource allocation result in penalties. The reward function is defined as:

$$R(s_t, a_t) = \sum_{i=1}^{n} w_i f_i(m_i) - P(a_t), \tag{15}$$

where:

- w_i represents the weight assigned to each performance metric, reflecting its relative importance in maintaining system stability and efficiency,
- $f_i(m_i)$ is a normalization function that scales metric values to a uniform range, ensuring comparability across heterogeneous indicators,
- $P(a_t)$ represents the penalty associated with a given action, discouraging unnecessary or inefficient resource allocation.

The total reward captures the trade-off between improvements in system performance and the cost associated with resource usage. Table 9 defines the reward distribution for different scenarios.

The reward values and corresponding weights were defined empirically through iterative experimentation, focusing on achieving stable convergence and balanced adaptation across workloads. Minor adjustments allowed to enhance the sensitivity of the agent with respect to latency and throughput, as these metrics exert a more direct influence on user experience and perceived quality of service. The resulting configuration exhibited consistent performance across multiple runs, confirming its reliability in promoting efficient resource utilization while maintaining overall system stability.

It is important to note that the adopted reward configuration is not static. The weights and reward magnitudes can be revalidated or adjusted according to the operational context, metric priorities, or hardware constraints of different environments. This flexibility allows Oraculum to generalize its reinforcement learning process while preserving the ability to fine-tune adaptation goals depending on deployment re-

Table 9
Reward and penalty values assigned to RL actions based on system performance impact.

Condition	Reward
CPU usage reduction by 3% (if above threshold)	+2.0
Memory usage reduction by 256MB (if above threshold)	+1.5
Decrease in response latency by 20ms	+3.0
Increase in throughput by 5%	+2.5
Reduction in packet loss by 0.5%	+1.8
Improved energy efficiency by 2%	+1.5
Increase in availability by 1%	+1.8
Increase in resilience by 1.5%	+1.5
CPU usage increase above optimal range	−2.0
Memory usage increase above optimal range	−1.5
Response latency increase by 20ms	−3.0
Throughput decrease by 5%	−2.5
Packet loss increase by 0.5%	−1.8
Availability reduction by 1%	−1.8
Resilience reduction by 1.5%	−1.5
Continuous scaling of CPU without performance gain	−4.5
Continuous scaling of memory without performance gain	−3.5

quirements and specific performance objectives. The penalty function $P(a_t)$ applies when scaling operations increase CPU or memory without observable performance improvements:

$$P(a_t) = \lambda_c \Delta CPU - \delta_c + \lambda_m \max(0, \Delta Mem - \delta_m), \tag{16}$$

where:

- λ_c and λ_m are penalty coefficients for CPU and memory usage,
- ΔCPU and ΔMem represent the change in resource allocation,
- δ_c and δ_m are predefined thresholds.

Each metric m_i is normalized and transformed by a function $f_i(m_i)$ tailored to its characteristics. For latency-related metrics, f_i is defined as an inverse function to penalize higher values, e.g., $f_i(m_i) = \frac{1}{1+m_i}$. For throughput or performance metrics, f_i is proportional to the metric value, e.g., $f_i(m_i) = \frac{m_i}{m_{\max}}$, where $m_{\max}$ is the maximum observed value. This design ensures the reward function appropriately balances conflicting objectives across heterogeneous metrics.

This formulation discourages unnecessary resource scaling, ensuring that actions optimize performance rather than simply consuming additional resources. The reward function continuously guides the agent towards balancing resource allocation, performance, and efficiency.

4.5. Problem formulation and parameter analysis

This section provides a comprehensive and explicit formulation of the optimization problem addressed by the Oraculum model, incorporating operational constraints and multi-objective considerations. It also presents an analysis of key configuration parameters and discusses how different workload characteristics affect model performance and reward stability.

Although the reinforcement learning (RL) objective in Eq. (11) is defined as maximizing the expected cumulative reward, the overall problem can be more clearly expressed as a constrained multi-objective optimization task. The goal is to optimize multiple system performance metrics, such as latency, SLA compliance, and energy consumption, while respecting operational limits that ensure system reliability and efficiency. Formally, the problem can be stated as:

$$\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right] \quad \text{subject to} \quad \begin{cases} \text{Latency}(s_t, a_t) \le L_{\max} \\ \text{SLA Violations}(s_t, a_t) \le V_{\max} \\ \text{Energy Consumption}(s_t, a_t) \le E_{\max} \end{cases}$$

where $L_{\max}$, $V_{\max}$, and $E_{\max}$ represent maximum allowable latency, SLA violation thresholds, and energy budgets, respectively. These constraints are critical to ensure that the system operates within acceptable performance and resource usage boundaries.

The reward function used in the model combines multiple objectives through a weighted summation of normalized metrics. However, this scalarization approach does not explicitly explore the Pareto front, the set of policies that optimally balance conflicting objectives. The absence of theoretical or empirical treatment of Pareto front exploration limits understanding of how the model balances trade-offs over time. Future work could incorporate multi-objective reinforcement learning techniques such as Pareto Q-learning or adaptive weight adjustment methods to enable dynamic balancing of objectives and approximation of Pareto-optimal policies under varying conditions.

The model's modular adaptation pipeline (Section 3.3) includes numerous configurable parameters ("knobs") that influence learning and adaptation behavior. An ablation study systematically varied key parameters and measured effects on convergence speed, stability, and final performance. Table 10 summarizes the parameters evaluated, their tested ranges, and observed impacts.

The ablation results indicate that reward weights are the primary factor in managing trade-offs among competing objectives, while learning rate and exploration parameters mainly affect training dynamics and stability. These findings highlight the importance of careful parameter tuning to achieve effective and stable adaptation.

Furthermore, the model was evaluated under different workload patterns, including bursty traffic and steady-state conditions. Bursty workloads introduce rapid and unpredictable changes in system state, which increase reward volatility and slow convergence due to frequent shifts in the environment. In contrast, steady-state workloads provide more stable conditions that facilitate smoother learning and policy refinement. These observations emphasize the need for workload-aware adaptation strategies that can dynamically adjust learning parameters or incorporate workload prediction to maintain performance and robustness across diverse operational scenarios.

These clarifications and analyses contribute to a deeper understanding of the Oraculum model's design, highlighting how operational constraints and multiple objectives are incorporated into the optimization process. The parameter sensitivity study underscores the importance of tuning key hyperparameters to balance performance trade-offs effectively. Additionally, the evaluation under diverse workload patterns reveals the model's adaptability and the challenges posed by dynamic environments, emphasizing the need for workload-aware strategies to maintain robust and efficient adaptation.

5. Results

This section reports the evaluation outcomes of the Oraculum architecture, with emphasis on adaptive behavior under varying workload conditions. The analysis includes metrics related to monitoring, prediction, and decision-making using the Twin-Delayed Deep Deterministic Policy Gradient (TD3) RL algorithm. Evaluation criteria include adaptation efficiency, resource utilization, and system stability (Dehghan Shoorkand et al., 2024; Yadav & Vishwakarma, 2023).

The experiments focus on three main aspects. First, adaptation-related metrics-MAT, AA, AO, and Adaptation Stability (AS)-are analyzed to quantify the performance of RL-based adjustments. Second, the architectural impact of adaptations is assessed across different layers of the system, including software, hardware, network, security, and service-level agreements (SLA). Third, the performance of the RL agent is examined in terms of action selection, reward accumulation, and decision quality across episodes.

The evaluation scenario involves controlled variation of workload intensities, gradually increasing system demand to observe the adaptation responses. The analysis quantifies how the architecture reallocates resources, mitigates performance degradation, and maintains operational thresholds under dynamic execution conditions. The following subsec-

Table 10
Ablation study of key configuration parameters.

Parameter	Tested Range	Impact on Performance
Learning Rate	$[10^{-5}, 10^{-3}]$	Controls convergence speed; high values cause instability, low values slow learning
Discount Factor (γ)	$[0.90, 0.999]$	Balances importance of immediate vs. future rewards; higher values favor long-term optimization
Reward Weights	Varied per metric	Most significant effect on balancing trade-offs between latency, energy, and SLA compliance
Batch Size	$[32, 256]$	Affects training stability and sample efficiency
Replay Buffer Size	$[10^4, 10^6]$	Larger buffers improve experience diversity but increase memory and computation
Exploration Parameters	Algorithm-dependent	Influence exploration-exploitation balance, affecting policy robustness and convergence

tions present a detailed breakdown of observed behaviors and corresponding performance measurements.

5.1. Performance evaluations

The prototype was executed in a containerized environment using Docker, hosted on a system equipped with an *Intel Core i5-12400* processor, 16 GB of RAM, and a 500 GB SSD. Resource limits (CPU and memory) were managed via Docker commands, enabling controlled stress conditions. The evaluation simulated increasing workloads and fault scenarios to assess the system's adaptive behavior (Gupta & Agarwal, 2025; Sobieraj & Kotyński, 2024).

To ensure realistic and representative workload dynamics, the evaluation relied on a combination of real-world IoT datasets and a controlled simulation environment. Specifically, it used three public IoT datasets–*SmartSantander*, *CityPulse*, and the *Intel Berkeley Research Lab*-together with the *SHiELD* simulator as complementary sources of workload generation:

- **SmartSantander:** It provides large-scale, real-world urban IoT data collected from thousands of sensors deployed across the city of Santander (Spain). It includes environmental variables (temperature, humidity, air pollution), mobility traces, and network latency indicators. Its high temporal resolution, heterogeneity, and open availability make it a well-recognized benchmark for validating smart-environment models under realistic conditions. It was used to evaluate predictive stability and context-adaptive decision-making under fluctuating urban workloads.
- **CityPulse:** It contains context-enriched data streams from multiple European cities, emphasizing events such as traffic congestion, environmental monitoring, and urban energy demand. It was selected to test Oraculum's capability to handle multi-source event fusion and dynamic context switching, supporting correlation between spatially distributed sensor nodes. This dataset validated Oraculum's semantic reasoning and event-classification components.
- **Intel Berkeley Research Lab:** It provides indoor environmental monitoring data captured from 54 wireless sensor nodes operating within a laboratory network. It includes temperature, humidity, light intensity, and voltage readings sampled every 30 s. Although smaller in scale, its high sampling frequency and noise variation are valuable for evaluating the precision of short-term forecasting models and anomaly detection.
- **SHiELD simulator:** It extends these datasets by generating scalable, fault-injectable workloads based on their statistical profiles. It allows controlled reproduction of sensor behaviors, missing data, delays, and performance degradation scenarios. The simulator is therefore instrumental for validating Oraculum under repeatable yet realistic stress conditions, bridging the gap between real-world heterogeneity and controlled testing environments.

Among these sources, *SmartSantander* served as the primary evaluation dataset, while *CityPulse* and *Intel Berkeley Research Lab* data allowed for calibrating and cross-validate predictive and semantic components. The *SHiELD* simulator acted as the workload orchestrator to ensure repeatability and reproducibility across experiments. This combination guarantees that the datasets collectively represent urban, industrial, and laboratory IoT settings, capturing both spatial and temporal diversity required for comprehensive adaptive-system evaluation.

To ensure transparency and reproducibility, the datasets used in the evaluation were fully characterized in terms of scale, variability, and statistical properties. The *SmartSantander* dataset comprises approximately 2.4 million records collected from more than 20,000 sensors distributed throughout the city of Santander (Spain), covering environmental and mobility parameters such as temperature, humidity, CO_2, NO_2, traffic intensity, and network latency. The dataset exhibits natural daily variation of up to 18% across measurements, with approximately 4.3% of missing values. These were treated using stochastic interpolation and Z-score normalization to ensure consistency with other sources.

The *CityPulse* dataset includes around 1.2 million multimodal events from six European cities, including IoT data on traffic, parking, energy use, and air quality. Roughly 12% of entries contain noise or incomplete context annotations, which were filtered using a dynamic thresholding approach based on variance deviation. The resulting dataset has medium temporal density and contextual diversity, making it ideal for validating semantic reasoning models.

The *Intel Berkeley Research Lab* dataset consists of about 2.3 million environmental samples captured from 54 wireless sensor nodes within a controlled indoor network. Measurements include temperature, humidity, light intensity, and voltage collected every 30 s. Its relatively uniform sampling rate and narrow signal variance (standard deviation $= 0.07$) allow fine-grained testing of Oraculum's anomaly-detection capability. All datasets were preprocessed using min-max normalization and aligned to a common timestamp resolution. The resulting unified data stream was then used by the *SHiELD* simulator to reproduce realistic workloads and controlled fault conditions.

The testing strategy introduced three workload levels (low, medium, and high) each with predefined conditions:

- **Low Load:** The system operated under minimal resource usage, with the SHiELD simulator generating lightweight periodic requests. Containers were constrained to a small fraction of CPU and memory (Morabito, 2017; Sun et al., 2024; Xu & Buyya, 2019).
- **Medium Load:** Additional concurrent processes simulated database access and API traffic. CPU and memory demand increased progressively, emulating real-world workload escalation. Docker resource limits were dynamically reconfigured to accommodate usage growth (Morabito, 2017; Raibulet et al., 2023).
- **High Load:** Stress operations introduced computational bottlenecks, including encryption, recursive calculations, and large in-memory datasets. Faults were injected by restarting services and introducing API delays. These actions tested the system's response to failures and performance degradation (Al-Sharafi et al., 2023; Velrajan & Sharmila, 2023).

Stress conditions were applied as follows:

- **CPU Stress:** High-intensity threads executed mathematical operations, controlled via:

```
docker update --cpus=X container                              (17)
```

- **Memory Stress:** Memory-bound tasks allocated persistent high-dimensional objects, and memory limits were configured using:

```
docker update --memory=Y container                            (18)
```

Table 11
Test scenarios and system load increase.

Plan	CPU (%)	Mem. (MB)	Dataset Size	Load Increase Method
Low Load	10–30	100–300	100,000 records	The SHiELD simulator generates periodic low-intensity requests. Containers are allocated 0.5 CPU cores and 200MB RAM to ensure low computational overhead.
Medium Load	40–70	400–700	200,000 records	The SHiELD simulator increases the number of requests proportionally. More concurrent database transactions and API calls are generated. CPU allocation is raised to 1.5 cores, and memory is increased to 600MB.
High Load	80–100	800–1200	400,000 records	CPU-intensive tasks such as recursive calculations, encryption, and high-frequency logging are executed. Some nodes are manually terminated to simulate failures, redistributing the load among the remaining nodes.

- **Failure Injection:** Critical services were restarted, nodes terminated, and artificial API delays inserted to simulate failures and network congestion.

Throughout the tests, the monitoring architecture collected metrics such as CPU and memory usage, response time, disk I/O, garbage collection frequency, and system availability. The strategy ensured repeatability and allowed observation of adaptive responses under escalating demands.

Table 11 summarizes the defined load levels and associated stress characteristics. The table provides an overview of the controlled increase in system stress and the corresponding methods applied to induce workload growth.

Figs. 5 and 6 illustrate the CPU and memory usage of each monitored service over a 30-min period. The graphs show how resource consumption evolves as the system transitions between different load levels.

The red vertical line represents the transition from low load to medium load at the 10-min mark. The blue vertical line indicates the transition from medium load to high load at the 20-min mark. These transitions mark moments when additional computational stress was applied, either by increasing concurrent requests, introducing heavier workloads, or simulating failures.

In Fig. 5, CPU usage remains low in the initial phase, with most services operating below 15%. As the load increases after the first transition, services such as the *Training-Scheduler* and *RL-Engine* show a noticeable increase in CPU consumption. This is expected since model training and RL require significant processing power. The *Trigger-Service* and *Prediction-Service* also experience a rise in CPU usage, reflecting the increased number of alerts being processed. By the high-load phase, CPU usage stabilizes at higher levels, with services such as the *RL-Agent* and *Action-Service* operating near their capacity limits.

Fig. 6 presents memory consumption trends. At low load, memory usage remains relatively stable, with minimal fluctuations. As the system transitions to medium load, services like the *Training-Scheduler* and *RL-Engine* start consuming more memory due to the increasing size of data batches and models being processed. The *Data-Collector* and *Prediction-Service* also show gradual memory growth, reflecting the accumulation of incoming data and predictions being generated. By the high-load phase, the *RL-Engine* reaches its peak memory usage, highlighting the computational demands of RL updates. Other services, such as the *Ontology Integration* and *API Collector*, experience moderate increases, while the *Action-Service* maintains a steady memory footprint.

The observed trends confirm the expected workload behavior under different load conditions. During the high-load phase, the heavy computational demands services exhibit the highest resource consumption, demonstrating the system's ability to allocate processing power and memory where needed. The results also indicate that resource utilization does not immediately spike at transition points but gradually adapts to the increased demand, reflecting a realistic workload progression.

Table 12 presents resource utilization metrics for each node under increasing workload levels. The results include CPU and memory usage, as well as execution times for component-specific operations. The analysis focuses on variations observed during the transition from low to high system load.

Table 12
Summary of monitored measurements for each node at different load levels.

Node Metric	Low	Medium	High
API Collector			
CPU Usage (%)	10.3–14.8	19.6–28.9	41.2–59.7
Memory Usage (MB)	152.7–248.3	257.4–487.2	505.9–793.5
Response Time (ms)	12.5–28.4	31.2–96.7	108.9–294.6
Ontology Integration			
CPU Usage (%)	4.9-9.7	14.2–23.8	32.5–49.1
Memory Usage (MB)	98.3–189.5	203.1–392.7	410.6–695.8
SWRL Query Execution Time (ms)	53.4–143.6	164.8–388.3	426.1–784.2
Dataset-Builder			
CPU Usage (%)	14.7–19.2	26.4–34.1	46.7–68.5
Memory Usage (MB)	207.5–289.2	312.6–587.9	621.4–892.3
Dataset Generation Time (s)	6.3–14.7	18.4–38.9	44.5–88.2
Training-Scheduler			
CPU Usage (%)	19.8–29.1	37.3–48.5	62.7–79.2
Memory Usage (MB)	314.5–496.2	512.7–798.3	845.9–1187.3
Model Training Time (s)	35.7–118.4	125.3–298.7	320.5–882.1
Data-Collector			
CPU Usage (%)	11.4–19.2	23.8–33.9	42.5–69.3
Memory Usage (MB)	153.2–292.6	317.5–589.8	635.7–889.4
Data Transfer Rate (Mbps)	12.6–28.5	34.2–76.8	85.9–147.4
Prediction-Service			
CPU Usage (%)	12.9–18.5	22.3–34.2	39.6–68.7
Memory Usage (MB)	186.5–342.1	357.8–684.9	718.6–998.4
Prediction Latency (ms)	21.3–48.7	55.2–144.9	162.3–396.5
Trigger-Service			
CPU Usage (%)	9.7–14.9	18.5–27.4	33.2–49.6
Memory Usage (MB)	204.8–289.6	314.3–487.1	526.7–695.9
Trigger Processing Time (ms)	11.4–28.6	33.7–95.2	113.8–247.3
	RL-Engine		
CPU Usage (%)	24.3–33.8	38.5–52.4	61.9–89.1
Memory Usage (MB)	414.6–589.2	618.7–876.1	954.3–1194.8
RL Model Decision Time (ms)	56.2–143.8	169.4–387.2	439.3–984.7
RL-Agent			
CPU Usage (%)	17.6–27.9	31.4–48.2	52.8–79.6
Memory Usage (MB)	362.1–487.5	523.6–789.4	846.7–1103.2
Action Selection Time (ms)	22.5–48.3	55.6–144.1	168.7–393.6
Action-Service			
CPU Usage (%)	13.8–21.6	27.9–38.7	41.6–68.3
Memory Usage (MB)	229.3–392.5	428.7–693.2	754.1–899.8
Action Processing Time (ms)	12.2–28.9	34.1–96.4	118.3–294.1

The *API Collector*, responsible for gathering metrics from multiple services, shows a steady rise in CPU and memory usage as the number of collected data points increases. Under high load, response times grow significantly due to the volume of requests being processed simultaneously. Since this component is responsible for maintaining a continuous data stream, any delay at this stage affects all subsequent processing steps.

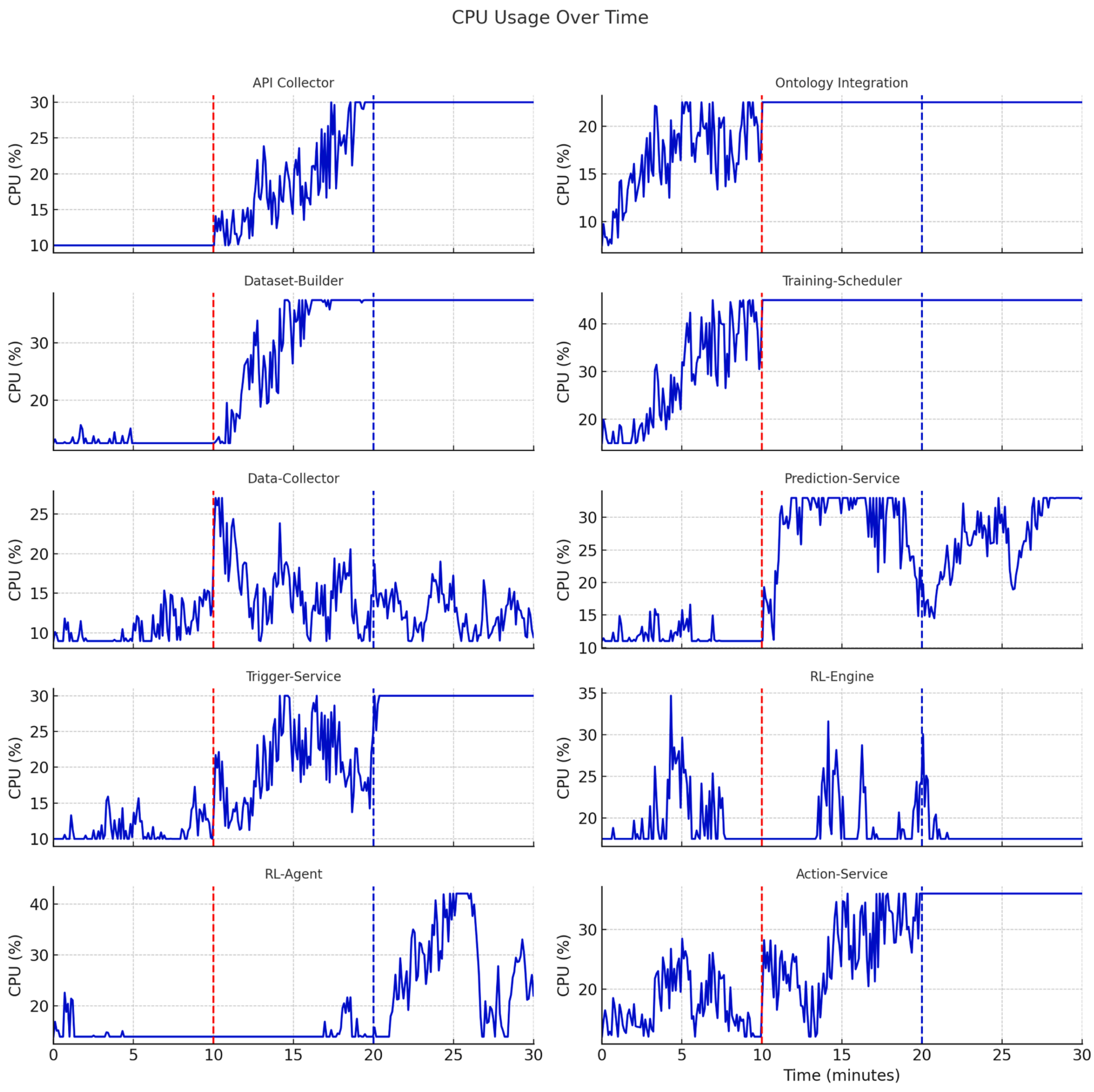

Fig. 5. CPU usage over time for monitored services.

Similar behavior is observed in the *Ontology Integration* service, which applies SWRL queries to classify instances based on real-time data. As the number of metrics collected increases, the query execution time also expands, requiring more processing resources. The complexity of reasoning grows as more instances are evaluated, leading to a noticeable increase in CPU and memory usage at high loads. This behavior highlights the computational cost of integrating real-time data into an ontology-based classification system.

The *Dataset-Builder* plays a critical role in structuring historical data for machine learning tasks. During low load, dataset generation remains relatively lightweight, but as more data points accumulate, the computational effort required to process and store them increases. The CPU and memory usage rise significantly in high-load scenarios, reflecting the higher volume of data being transformed and saved. The dataset generation time also escalates, indicating that larger datasets require proportionally more resources to be structured and formatted.

The *Training-Scheduler*, responsible for the construction of predictive models, exhibits one of a particularly noticeable increase in resource consumption. As more data is fed into the system, model training becomes computationally expensive, with processing times increasing from seconds to several minutes. The CPU and memory demand intensifies under high load as the system attempts to optimize model parameters and evaluate different learning algorithms. This component is particularly affected by workload variations, as the efficiency of the entire predictive model depends on its ability to train models effectively.

Once models are trained, the *Prediction-Service* performs real-time inference to forecast future system behavior. As the volume and complexity of the incoming data increases, the usage of CPU and memory increases, leading to an increase in the prediction latency. This latency is critical, as delayed predictions can reduce the effectiveness of proactive system adjustments. The high-load scenario reveals how inference operations become progressively slower, emphasizing the need for computational efficiency when making real-time predictions.

The *Data-Collector* continuously transfers metrics from monitored components to the prediction pipeline. As workload increases, data transfer rates escalate, leading to higher CPU and memory usage. The rising throughput under high load places additional pressure on the system, as more resources are required to handle incoming data streams

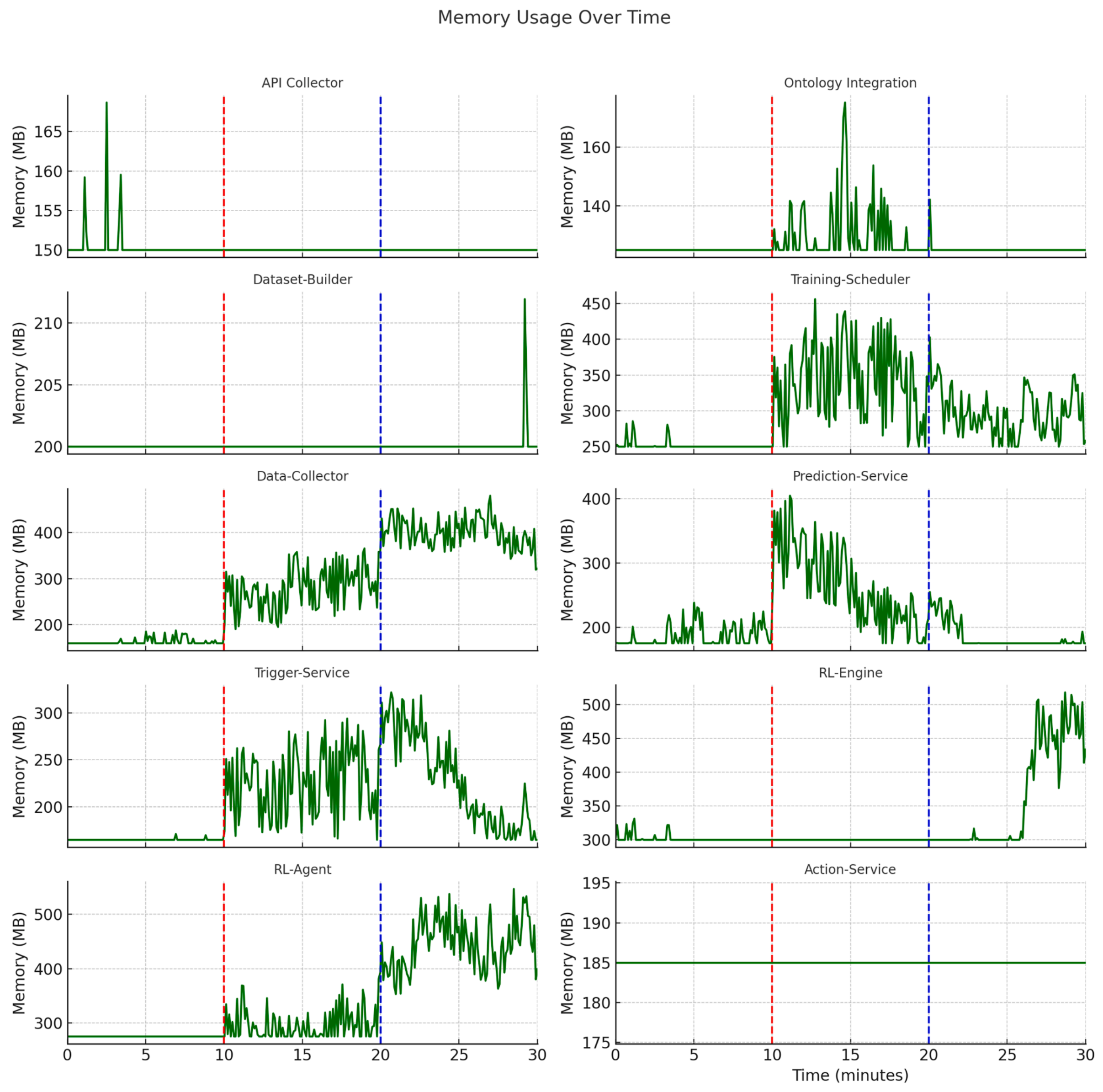

Fig. 6. Memory usage over time for monitored services.

while maintaining responsiveness. The efficiency of data transfer directly impacts downstream services, particularly those involved in real-time inference and decision-making (Hameed et al., 2021).

When anomalies are detected, the *Trigger-Service* initiates corrective actions based on predefined policies. This service shows a gradual increase in resource consumption, with trigger processing times growing as more alerts are generated. Under high load, the increased number of anomaly detections leads to a higher processing burden, reinforcing the importance of maintaining a balance between detection speed and action execution.

The *RL-Engine* is responsible for dynamically adjusting system parameters through RL. As system complexity grows, the RL model needs to evaluate a broader set of states and potential actions, increasing its computational demand. The decision time expands under high load, reflecting the increased effort required to determine optimal system adjustments. This behavior indicates that RL-based adaptation introduces additional latency, which must be managed to ensure timely responses.

Once an action is selected, the *RL-Agent* executes the necessary system adjustments. Under low load, decisions are infrequent, leading to minimal resource consumption. However, as anomalies become more common, the agent must evaluate system conditions more frequently, leading to increased CPU and memory usage. The action selection time grows significantly under high load, reflecting the complexity of determining the best response given multiple concurrent system changes.

Finally, the *Action-Service* applies the selected adjustments and validates system modifications. At low load, actions are rare, keeping processing times minimal. As workload increases, a higher number of corrective actions is required, leading to greater resource usage. The execution time for each action grows under high load, emphasizing the impact of frequent system modifications on overall stability.

These results highlight how each component of the monitoring architecture scales with increasing demand. CPU and memory consumption rise proportionally to workload intensity, and processing times reflect the computational cost of handling larger data volumes. Services directly involved in real-time decision-making, such as the Prediction-Service and RL-Engine, exhibit particularly high sensitivity to increased loads. The structured evaluation of system behavior under different conditions ensures that performance bottlenecks and potential optimization areas are identified.

Table 13
Evaluation metrics summary.

Metric	Formula	Interpretation
Core Metric - Geometric Evaluation		
Polygon Area Metric (PAM)	$\frac{1}{2}\sum_{i=1}^{n-1} \lvert(y_i-\hat{y}_i)(t_{i+1}-t_i)\rvert$	Measures geometric deviation between predicted and actual values, capturing overall trend differences.
Regression Metrics - Statistical Evaluation		
Mean Normalized Bias (MNB)	$\frac{1}{n}\sum_{i=1}^{n}\frac{y_i-\hat{y}_i}{y_i}$	Identifies systematic bias in predictions.
Mean Absolute Scaled Error (MASE)	$\frac{\sum_{i=1}^{n}\lvert y_i-\hat{y}_i\rvert}{\frac{1}{n-1}\sum_{i=2}^{n}\lvert y_i-y_{i-1}\rvert}$	Compares performance against a naïve baseline.
Normalized RMSE (NRMSE)	$\frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}}{y_{\max}-y_{\min}}$	Normalized measure of prediction error.
Adjusted R^2	$1-\left(\frac{(1-R^2)(n-1)}{n-p-1}\right)$	Adjusts R^2 for the number of predictors, preventing overfitting.
Nash-Sutcliffe Efficiency (NSE)	$1-\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}$	Evaluates predictive accuracy relative to data variability.
Classification Metrics - Model Discrimination		
Accuracy (CA)	$\frac{TP+TN}{TP+TN+FP+FN}$	Proportion of correctly classified instances.
Sensitivity (SE)	$\frac{TP}{TP+FN}$	Proportion of correctly identified positive instances.
Specificity (SP)	$\frac{TN}{TN+FP}$	Proportion of correctly identified negative instances.
AUC	—	Measures the ability to rank positive instances higher than negative ones.
F1 Score	$2\times\frac{Precision\times Recall}{Precision+Recall}$	Balances precision and recall in imbalanced datasets.

5.2. *Prediction results*

Evaluation of model performance in both regression and classification tasks uses a combination of geometric and statistical metrics. The Polygon Area Metric (PAM) serves as the central metric, measuring geometric deviation between predicted and actual time series values. Unlike traditional statistical metrics, PAM captures overall trend differences rather than isolated errors, making it suitable for time-dependent predictions. The formula for PAM is:

$$PAM = \frac{1}{2}\sum_{i=1}^{n-1}\left|(y_i-\hat{y}_i)(t_{i+1}-t_i)\right| \tag{19}$$

where y_i and $\hat{y}_i$ are the actual and predicted values at time i, t_i is the corresponding time step, and n is the total number of observations (Aydemir, 2021).

Since PAM alone does not capture all aspects of model accuracy, additional statistical metrics were used to complement its evaluation. In regression tasks, Mean Normalized Bias (MNB) detects systematic over- or under-estimations, Mean Absolute Scaled Error (MASE) compares the model to a naïve baseline, and Normalized Root Mean Squared Error (NRMSE) standardizes errors across different scales. Adjusted R-squared (R^2_{adj}) prevents overestimation of model fit, while Nash-Sutcliffe Efficiency (NSE) evaluates predictive performance against observed variance.

For classification tasks, Accuracy (CA), Sensitivity (SE), and Specificity (SP) measure correct classifications, while Area Under the Curve (AUC) assesses the model's ability to distinguish between classes. The F1 Score balances precision and recall, ensuring reliable evaluation in imbalanced datasets. Table 13 summarizes the applied metrics, their formulas, and interpretations.

The evaluation model provides a structured assessment of model accuracy across different learning tasks by combining PAM with statistical and classification metrics. PAM ensures the detection of long-term trends, while statistical metrics refine the analysis of prediction errors and classification quality. Fig. 7 shows the performance of regression models across CPU, memory, GPU, and storage. The linear regression model exhibited stable performance but struggled in scenarios where relationships between variables were nonlinear. The decision tree regressor captured more complex patterns but was sensitive to overfitting, leading to varying performance across different metrics. The neural network models demonstrated higher adaptability, particularly NN2 and NN3, which incorporated additional layers and dropout to enhance generalization. NN3 achieved more balanced results, especially in CPU and storage metrics, indicating that regularization helped mitigate overfitting. The results suggest that deeper architectures performed better when capturing intricate relationships in the dataset, while simpler models remained competitive in cases where patterns were less complex.

Fig. 8 illustrates the performance of the classification models. Decision tree and logistic regression models provided competitive results, particularly in CPU and storage, where decision boundaries were well-defined. Neural networks demonstrated improvements in memory and GPU-related metrics, where feature extraction played a more significant role in distinguishing classes. The deeper structures of NN2 and NN3 provided enhanced decision-making capabilities by capturing hierarchical relationships in the data. However, the increased complexity of NN3 did not always translate into better performance, as its gains were marginal compared to NN2 in certain cases. Logistic regression showed stable results, particularly in structured datasets with clearer separability. The decision tree classifier achieved reasonable performance but exhibited variations due to its sensitivity to small fluctuations in the dataset.

The results highlight how model complexity influences predictive accuracy. Neural networks captured deeper relationships in complex datasets, while simpler models such as decision trees and logistic regression remained effective when patterns were well-defined. The selection of an appropriate model depends on the dataset characteristics, balancing interpretability, computational cost, and adaptability to varying conditions.

In the broader evaluation, the analysis of software and network metrics reinforced the observed trends. For software performance, deeper neural networks (NN2 and NN3) consistently outperformed linear regression and decision tree models, especially in metrics sensitive to non-linear relationships and generalization, such as NRMSE and NSE. For example, NN3 achieved an NRMSE as low as 0.12 and an NSE of 0.91 in latency prediction tasks, while linear regression and decision trees exhibited higher error rates and lower stability, with NRMSE values above 0.20 in more complex scenarios. Logistic regression maintained stable results in throughput classification, with F1-scores around 0.81, but was surpassed by neural networks, which reached F1-scores above 0.88 in several tasks. The use of dropout in NN3 further improved generalization, particularly in garbage collection time and throughput predictions.

Similarly, for network-related metrics, the regression and classification results confirmed the advantage of deep learning approaches. NN2 and NN3 achieved the lowest prediction errors and highest classification scores, with NN2 reaching an NRMSE of 0.10 and an AUC of 0.93 in packet loss prediction. Decision trees and logistic regression models remained competitive in well-structured cases but struggled with recall and sensitivity in more variable conditions, with sensitivity values dropping below 0.75 in some packet loss scenarios. The improvements ob-

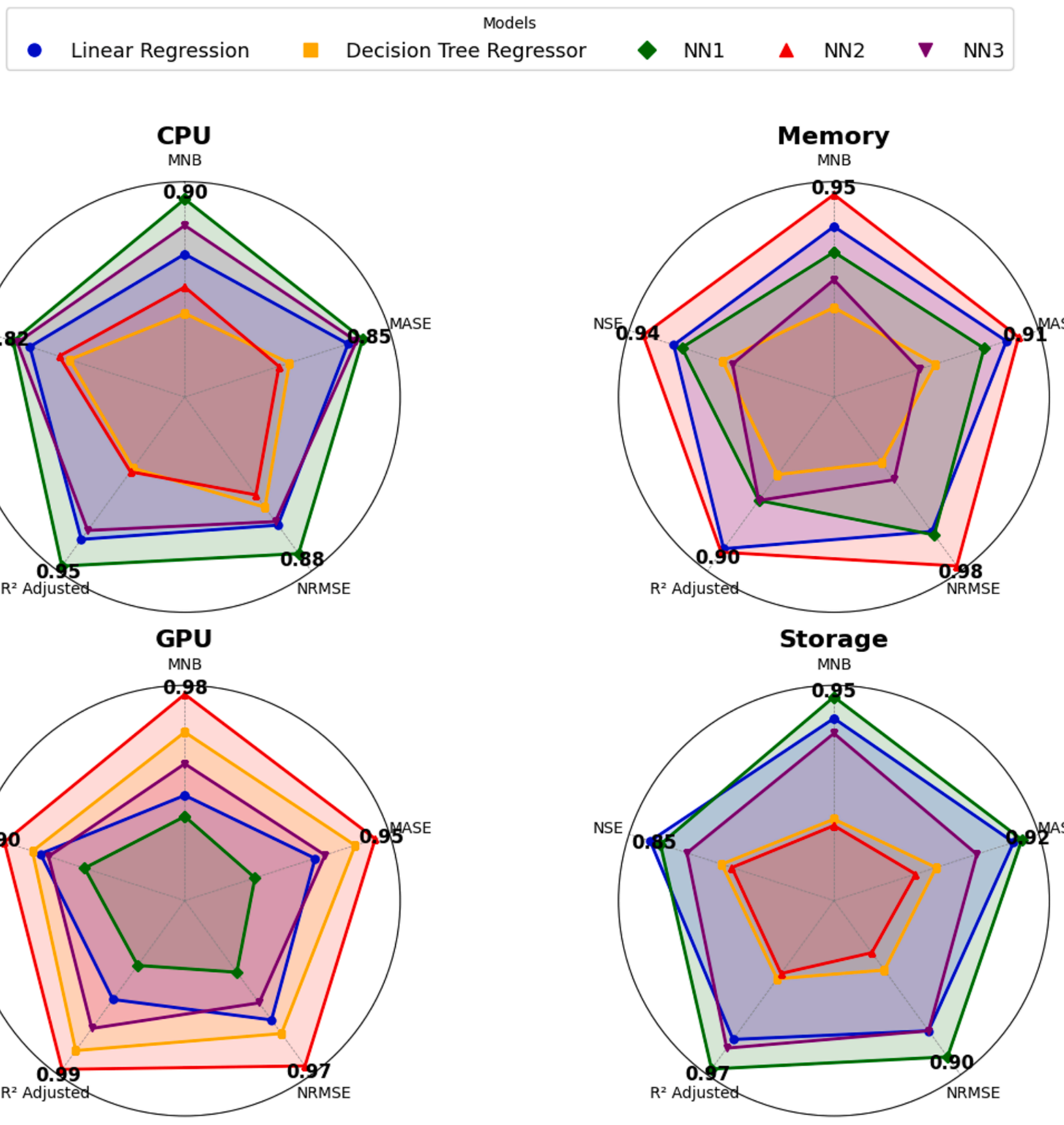

Fig. 7. Regression model performance across hardware-related metrics.

served with deeper neural networks highlight the importance of model complexity for capturing the multifaceted nature of both software and network performance, while also emphasizing the need to balance complexity with interpretability and computational efficiency depending on the application context. The complete set of results, including detailed tables and additional metrics for all experiments, is available in the project's GitHub repository.[3]

Table 14 summarizes the average evaluation metrics obtained for both anomaly detection and regression tasks. These metrics provide a detailed view of the system's performance in correctly identifying anomalies and accurately predicting system behavior.

The false positive rate (FPR) and false negative rate (FNR) are particularly important for understanding the practical impact of the model. A low FPR (5.2% for classification and 4.8% for regression) indicates that the system rarely triggers unnecessary adaptations, which helps avoid wasted resources and potential performance degradation caused by false alarms. Conversely, the FNR (7.1% for classification and 6.5% for regression) reflects the proportion of true anomalies that the system fails to detect, representing a risk of missed critical events that could lead to system instability or SLA violations. Balancing these rates is important to ensure reliable and efficient adaptation, and the reported values demonstrate that Oraculum maintains this balance effectively.

5.3. RL agent results

This section presents the performance evaluation of the proposed architecture, incorporating key adaptation metrics to assess its effectiveness. The evaluation includes MAT, AA, AO, and adaptation stability (AS). These metrics quantify how efficiently and effectively the system responds to environmental changes and adaptive actions. The MAT measures the elapsed time between anomaly detection and completed adaptation:

$$MAT = \frac{\sum_{i=1}^{n}(t_{\text{end-adaptation}} - t_{\text{detection}})}{n} \tag{20}$$

The AA quantifies the percentage of successful adaptations that effectively mitigate detected issues:

$$AA = \frac{\text{Number of successful adaptations}}{\text{Total number of adaptations performed}} \times 100 \tag{21}$$

[3] https://github.com/DarlanNoetzold/Oraculum-Model

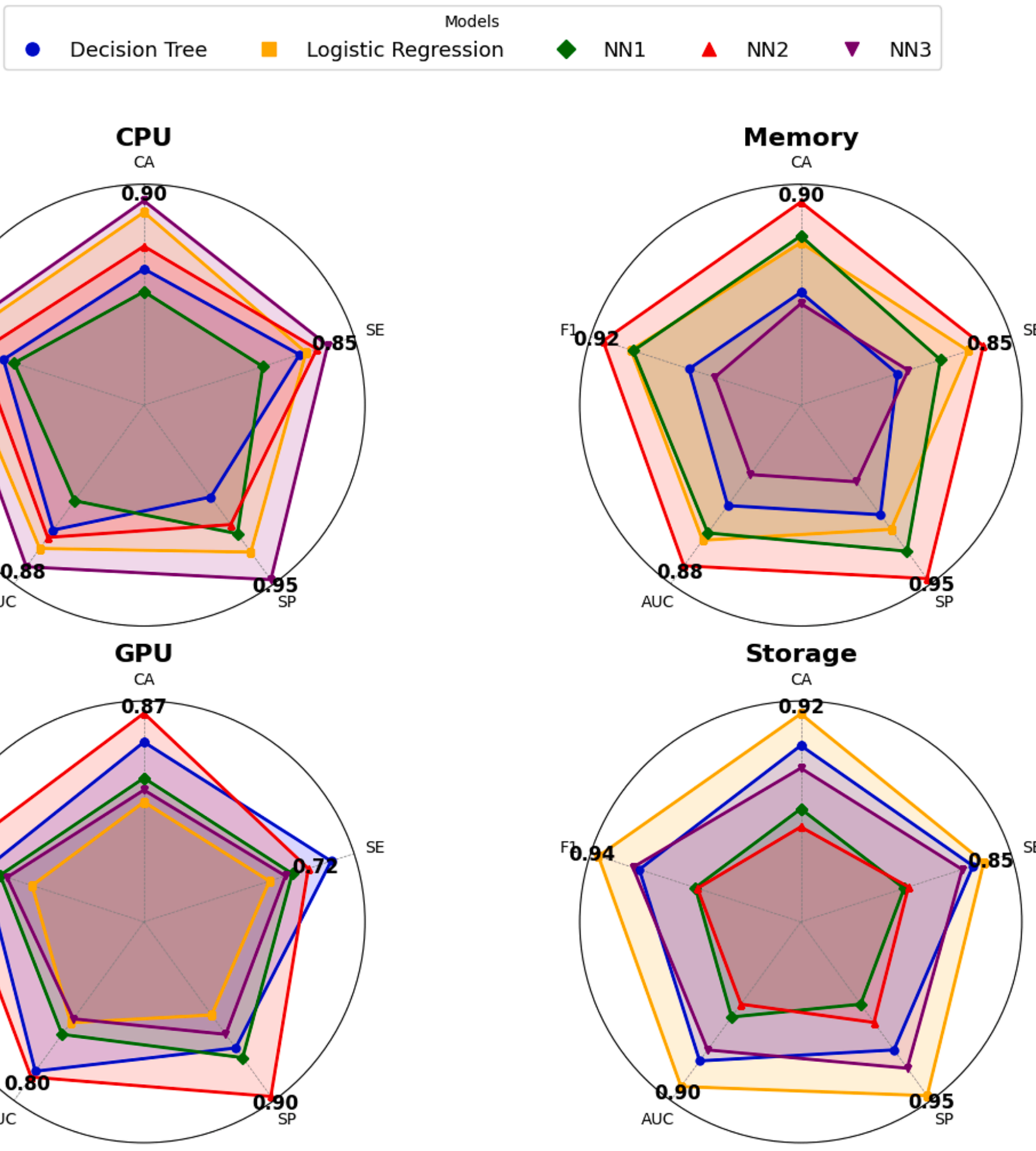

Fig. 8. Classification model performance across hardware-related metrics.

Table 14
Summary of evaluation metrics for anomaly detection and regression tasks (average values).

Metric	Value	Description
Precision	87.5%	Proportion of correctly identified positive instances
Recall (Sensitivity)	82.3%	Proportion of actual positives correctly identified
F1-score	84.8%	Harmonic mean of precision and recall
False Positive Rate (FPR)	5.2%	Proportion of negatives incorrectly classified as positive
False Negative Rate (FNR)	7.1%	Proportion of positives incorrectly classified as negative
Accuracy (CA)	90.2%	Overall proportion of correct classifications
Specificity (SP)	94.8%	Proportion of actual negatives correctly identified
AUC	0.91	Area under the ROC curve, measure of ranking quality
Polygon Area Metric (PAM)	0.035	Geometric deviation between predicted and actual values
Mean Normalized Bias (MNB)	0.012	Systematic bias in predictions
Mean Absolute Scaled Error (MASE)	0.85	Error relative to naïve baseline
Normalized RMSE (NRMSE)	0.10	Normalized root mean squared error
Adjusted R^2	0.87	Goodness of fit adjusted for predictors
Nash-Sutcliffe Efficiency (NSE)	0.82	Predictive accuracy relative to data variance
False Positive Rate (FPR) (Regression)	4.8%	Rate of false alarms in regression residuals
False Negative Rate (FNR) (Regression)	6.5%	Rate of missed anomalies in regression residuals

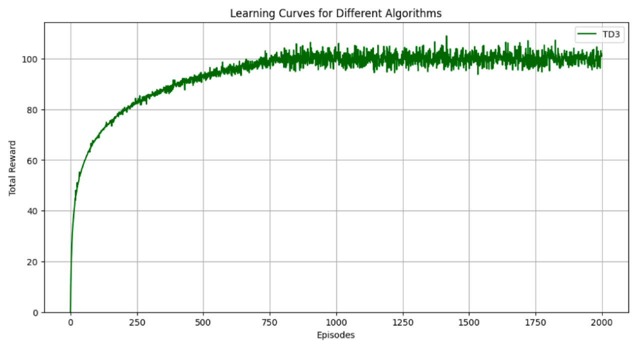

Fig. 9. Learning curve of TD3, the best-performing RL model with rapid convergence and stable rewards.

The AO evaluates the additional resource consumption introduced by the adaptation process:

$$AO = \frac{\text{Resource usage after} - \text{Resource usage before}}{\text{Resource usage before}} \times 100 \tag{22}$$

Finally, adaptation stability (AS) assesses how stable system performance remains after adaptation:

$$AS = \frac{1}{\sigma^2_{\text{performance post-adaptation}}} \tag{23}$$

The proposed Oraculum achieved a very low MAT of 0.05 s, high AA of 97%, minimal AO of 2%, and system stability reaching 98%. The learning process of RL models was evaluated using Q-learning, PPO, SAC, and TD3 over 2000 episodes. Each model demonstrated different levels of convergence, stability, and policy effectiveness.

Identical experimental conditions (same environment dynamics, reward function, observation and action spaces, and random seeds) ensured a fair comparison among RL algorithms during training and evaluation. Hyperparameters were systematically tuned for each algorithm using a two-stage procedure. First, a coarse grid search explored learning rate, discount factor γ, and target-update related parameters within ranges commonly reported in the literature for continuous control tasks. Second, a finer local search refined the best-performing region for each algorithm. The following hyperparameter ranges were considered for all actor-critic methods (PPO, SAC, TD3): learning rate in $\{1\times10^{-5}, 3\times10^{-5}, 1\times10^{-4}, 3\times10^{-4}, 1\times10^{-3}\}$, discount factor $\gamma \in \{0.95, 0.97, 0.99\}$, and batch size in $\{128, 256, 512\}$. Additional algorithm-specific parameters were tuned within standard ranges: for PPO, clipping coefficient $\in \{0.1, 0.2, 0.3\}$ and update epochs $\in \{5, 10\}$; for SAC, target entropy scaling and temperature learning rate; for TD3, policy delay $\in \{1, 2\}$ and target policy noise standard deviation $\in \{0.1, 0.2, 0.3\}$. Q-learning used a discrete action approximation and followed a dedicated grid search for learning rate and γ in the same ranges. For each configuration, performance was averaged over 30 independent runs to mitigate sensitivity to random initialization.

Among the reinforcement learning algorithms evaluated, TD3 achieved the best overall performance. As shown in Fig. 9, TD3 exhibited the highest reward accumulation and the lowest variance across training episodes. Its twin-delayed update mechanism and target policy smoothing contributed to rapid convergence and robust policy stability, efficiently mitigating overestimation bias and preventing abrupt policy changes. The learning curve demonstrates a swift increase in reward followed by consistent convergence, indicating that TD3 effectively identified optimal adaptation actions in the Oraculum environment.

The comparative analysis across algorithms considered both convergence speed and stability. Convergence speed was measured as the number of episodes required to reach 90% of the maximum average reward observed for that algorithm. Stability was measured as the inverse of the reward variance over the last 500 training episodes. Table 15 summarizes the mean and standard deviation of these metrics over 30 runs.

Table 15
Convergence speed and stability statistics (mean ± standard deviation over 30 runs).

Algorithm	Episodes to 90% max reward	Reward variance
Q-learning	> 2000 (no stable convergence)	high, unstable
PPO	643 ± 61	0.182 ± 0.024
SAC	587 ± 52	0.149 ± 0.019
TD3	$\mathbf{412 \pm 37}$	$\mathbf{0.096 \pm 0.015}$

TD3 converged in 412 ± 37 episodes, compared to 587 ± 52 for SAC and 643 ± 61 for PPO, and achieved the highest stability with the lowest reward variance in the late training phase.

Pairwise comparisons between TD3 and the other algorithms used two-sample t-tests over the 30 independent runs to validate these differences statistically. For convergence speed, TD3 significantly outperformed PPO and SAC (TD3 vs. PPO: $p = .003$; TD3 vs. SAC: $p = .011$). For stability, measured as the variance of rewards in the last 500 episodes, TD3 also showed statistically lower variance (TD3 vs. PPO: $p = .007$; TD3 vs. SAC: $p = .019$). All p-values were below the 0.05 threshold, indicating statistically significant improvements in both convergence speed and stability. Q-learning exhibited substantially slower convergence and higher variance, and therefore was not competitive in this continuous control setting.

Given these results, TD3 was selected as the reinforcement learning algorithm for all subsequent Oraculum experiments and evaluations. The learning curves and detailed results for Q-learning, PPO, and SAC, including their respective limitations in convergence speed, reward stability, and peak performance, are available in the project's GitHub repository[4]. This choice ensures that the adaptation strategies employed by Oraculum are both efficient and robust, leveraging the strengths of TD3 in complex, dynamic environments.

A key rationale for selecting TD3 as the primary RL method in Oraculum lies in its ability to address common challenges faced by other algorithms in continuous control tasks. Unlike Q-learning, which is limited to discrete action spaces and often struggles with convergence in complex environments, TD3 operates effectively in continuous action domains. Compared to PPO, which can experience reward fluctuations due to its clipped objective, TD3 employs twin-delayed updates and target policy smoothing to stabilize training and reduce overestimation bias. While SAC also provides strong stability through entropy regularization and automatic tuning of exploration, TD3's deterministic policy and delayed updates enable it to learn more precise and efficient adaptation strategies. This combination results in faster convergence and higher accumulated rewards, as reflected in Tables 16 and 15, making TD3 a suitable choice for optimizing system performance in dynamic and heterogeneous smart environments.

The system's operational performance was assessed across multiple categories, including software, hardware, network, security, and SLA metrics. Figs. 10–13 present the results, highlighting how the architecture responded to different workload conditions and adaptation processes.

Fig. 10 illustrates the trends in software performance metrics over time. Throughput increased steadily, indicating that the system handled a growing number of requests while maintaining processing capacity. Response latency and garbage collection time exhibited a downward trend, suggesting improvements in task scheduling and memory management. These trends indicate that the RL agent's adaptive strategies contributed to better workload distribution and resource allocation.

The gradual decrease in response latency suggests that adjustments in CPU and memory allocation reduced bottlenecks and improved overall system responsiveness. The increasing throughput demonstrates that the system sustained a consistent processing rate despite variations in

[4] https://github.com/DarlanNoetzold/Oraculum-Model

Table 16
Count and reward for each action across models, reflecting variations in RL decision-making.

Action	Q-learning		PPO		SAC		TD3	
	Count	Reward	Count	Reward	Count	Reward	Count	Reward
scale up cpu	917	−318.50	1348	4148.75	792	4998.50	803	6397.25
scale up memory	148	−218.75	1395	3602.30	983	5321.45	1347	−6153.20
scale up gpu	1047	397.50	623	4998.75	1583	−5532.50	123	6351.75
scale up storage	977	−498.25	498	4201.80	1498	5202.35	583	6663.10
scale down cpu	298	347.25	523	4702.45	303	5298.75	1203	6801.50
scale down memory	703	−247.80	693	4302.60	183	4901.25	1698	7702.75
scale down gpu	113	382.25	178	4798.50	653	4312.75	733	6302.40
scale down storage	503	-268.90	583	4602.35	323	5501.80	1697	6663.15
restart node	463	−387.50	478	3998.75	793	5432.25	1398	7173.50
increase cpu	663	−458.20	898	4701.60	1233	6502.45	1647	7113.75
increase memory	703	−228.75	1398	4401.85	1083	5102.10	1103	6661.90
adjust filter	643	−297.50	598	4912.30	453	4621.75	1008	6303.25
adjust aggregation	1098	−388.90	283	3102.45	1498	−5603.50	1703	6402.10
adjust compression	83	422.50	1498	4113.25	923	5801.80	653	6293.75

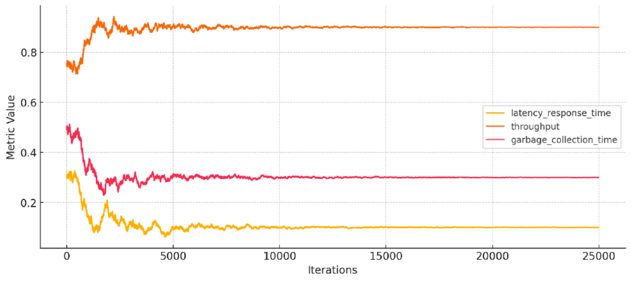

Fig. 10. Evaluation of software performance metrics.

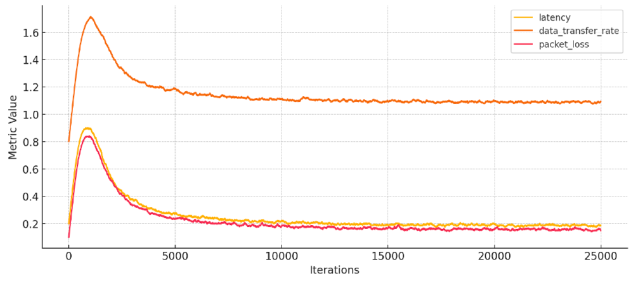

Fig. 12. Analysis of network performance metrics.

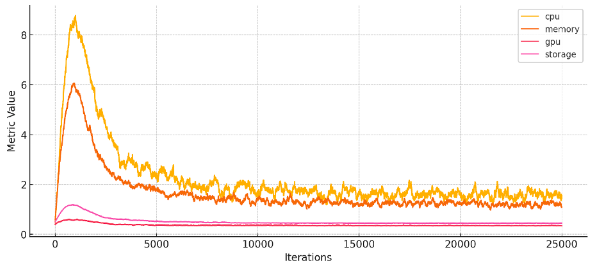

Fig. 11. Evaluation of hardware resource utilization.

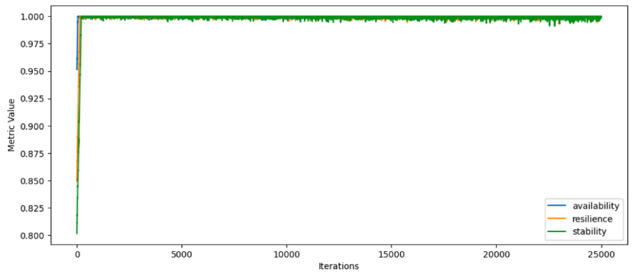

Fig. 13. Analysis of service level metrics.

workload. The declining garbage collection time indicates that memory management mechanisms, such as heap allocation and object recycling, functioned efficiently, reducing the need for frequent and prolonged garbage collection cycles.

Fig. 11 presents the utilization trends for CPU, memory, GPU, and storage throughout the monitored period. CPU and memory usage increased significantly at the beginning, likely due to system initialization and workload adaptation. As the system adjusted its resource allocation, both metrics stabilized, suggesting that the RL agent effectively optimized hardware utilization over time.

GPU usage remained consistently low, indicating that the computational workload relied primarily on CPU processing rather than GPU-accelerated tasks. This pattern suggests that the system’s adaptive strategies focused on CPU-bound operations, which aligns with the nature of the workload being executed. Storage utilization exhibited minimal variation, with low overall usage. The absence of significant fluctuations suggests that the system did not require extensive disk operations, indicating that caching and memory-based processing were sufficient to handle data requirements.

Fig. 12 presents the variations in network performance metrics, including latency, data transfer rate, and packet loss, across multiple monitoring iterations. Initially, latency increased sharply, likely due to fluctuations in resource allocation and network congestion. As the system adjusted to workload demands, latency gradually decreased and reached a stable level, indicating improved data transmission efficiency.

Packet loss followed a similar pattern, increasing in the early iterations before stabilizing at a lower level. This trend suggests that initial network adaptation led to temporary inconsistencies in data transmission. Over time, the system optimized network configurations, leading to more reliable packet delivery and reduced transmission errors. The data transfer rate increased during the adaptation phase and remained stable at a high level. This indicates that the system maintained efficient bandwidth utilization while minimizing disruptions caused by latency and packet loss.

Fig. 13 presents the evaluation of service-level metrics, including availability, resilience, and stability. Availability exhibited periodic fluctuations, reflecting system responses to workload variations and resource reallocation. Peaks in availability corresponded to periods of improved resource distribution, while occasional declines indicated instances where the system struggled to meet optimal service levels due to temporary constraints.

Resilience followed a similar trend, with variations influenced by system adaptation mechanisms. The results suggest that resilience adjustments occurred dynamically in response to shifts in performance demands. Spikes in resilience values indicate corrective measures applied to mitigate temporary inefficiencies, while drops suggest moments of increased system stress or adaptation delays.

Stability remained consistently high across iterations, suggesting that system reconfigurations did not introduce significant disruptions. While availability and resilience fluctuated based on operational conditions, stability measurements indicate that the system maintained predictable and controlled behavior over time.

Overall, the evaluation highlights the impact of RL-based adaptation in maintaining system performance. The tests conducted with TD3 demonstrated its ability to select effective actions, contributing to efficient resource allocation and minimal AO. The system successfully adjusted to workload variations, ensuring stable performance while maintaining high availability and resilience. The results indicate that the proposed architecture can adapt dynamically to changing operational conditions, optimizing performance with controlled resource consumption.

5.4. Evaluation with public benchmarks

To avoid overfitting the evaluation to a single simulated setting, the validation also considered three public datasets (or benchmarks) and three representative reference architectures. The datasets were: (i) (Sanchez & Galache, 2024) for smart-city conditions; (ii) (Luo et al., 2016) for mobility and traffic aware systems; and (iii) (Lab, 2004) for sensor-based adaptive platforms. These benchmarks supported a comparison of Oraculum against three architectures that target similar scenarios: (Tam et al., 2022), (Velrajan & Sharmila, 2023), and (Liu et al., 2021).

Each reinforcement learning configuration (TD3, SAC, and PPO) ran under identical test conditions to ensure reproducibility. All experiments used 30 independent runs, each consisting of 5,000 training episodes and 1,000 evaluation episodes. Each run used a distinct random seed initialization for the neural networks to mitigate random bias, with a batch size of 256 and a replay buffer of 500,000 transitions. During analysis, MAT, AA, AO, and Stability were computed as the mean values across the 30 runs, and the corresponding standard deviations were recorded internally for reproducibility verification.

To validate the statistical significance of the reported metrics, a one-way analysis of variance (ANOVA) at a 95% confidence level evaluated the effect of the architecture on MAT, AA, AO, and Stability for each benchmark. Tukey's HSD post-hoc tests then produced pairwise comparisons between Oraculum (TD3) and the baselines (SAC, PPO, and the three reference architectures). For instance, on the SmartSantander dataset, Oraculum obtained a MAT of 0.073 ± 0.011 s, AA of $95.2 \pm 1.4\%$, AO of $3.8 \pm 0.9\%$, and Stability of $97.5 \pm 1.2\%$ across the 30 runs, while the best-performing baseline (Velrajan et al. 2023) reached a MAT of 1.486 ± 0.137 s and Stability of $90.7 \pm 2.1\%$. The ANOVA indicated a significant effect of the architecture on MAT and Stability ($p < .001$ in both cases), and Tukey's tests showed statistically significant differences between Oraculum and the baselines for MAT (e.g., $p = .0027$ against Velrajan et al. 2023) and Stability (e.g., $p = .0084$ against Tam et al. 2022). Similar patterns appeared in the CityPulse and Intel Lab Data benchmarks, with p-values below 0.05 for the main adaptation metrics. These results excluded random variation as the primary explanation for the observed improvements and attributed the gains to the algorithmic characteristics of TD3, particularly its twin critic networks and delayed policy updates, combined with the orchestration logic of Oraculum's adaptation loop. Detailed per-run statistics and additional plots for all metrics and datasets remain available in the project's GitHub repository.[5]

All models executed within the SHiELD simulation model, extended to support external datasets and controlled injection of workload variations. The testbed enforced equivalent conditions for adaptation triggers, metric forecasting, and action validation across all models. Only these three reference architectures entered the benchmark due to the feasibility of reproducing their designs and adaptation strategies with the available implementation resources.

All applications executed on the SHiELD simulation model, extended to support ingestion of real-world datasets and controlled workload injection. Each reference model was reimplemented and integrated into this environment, which enabled uniform measurement of adaptation behavior, response time, prediction accuracy, and resource usage. The experimental setup included modules for event triggering, data transformation, and adaptive response generation, monitored using Prometheus and Grafana to ensure consistent logging of system dynamics.

The benchmark datasets were integrated according to the original specifications provided by their respective authors. The SmartSantander dataset (Sanchez & Galache, 2024) contained real-time sensor data from urban infrastructure such as lighting, temperature, and traffic sensors; this dataset supported the simulation of city-scale performance degradation and adaptive responses. The CityPulse dataset (Luo et al., 2016) comprised temporally correlated mobility and traffic event streams from urban environments, which were mapped to system metrics such as latency, throughput, and SLA compliance. The Intel Lab Data (Lab, 2004) provided dense time-series sensor data (e.g., temperature, humidity, and light intensity) captured in a wireless sensor network context, allowing the evaluation of resource-aware adaptation under environmental fluctuations. Each dataset was preprocessed to fit a unified format and injected into SHiELD using controlled timing and fault patterns, which ensured reproducibility and comparability across models.

The evaluated performance indicators included adaptation time, accuracy, overhead, prediction error, classification quality, and latency improvements. Table 17 summarizes the mean values of these indicators across the 30 independent runs for each model and benchmark.

The results indicated that Oraculum achieved lower adaptation times (MAT), higher AA, and increased system stability across the evaluated benchmarks. These outcomes pointed to an effective use of the reinforcement learning component for generating timely and context-aware adaptation decisions. The RMSE values remained lower in all testbeds, which suggested consistent performance of the regression models in forecasting system metrics, while the F1-scores indicated adequate behavior of the classification module in detecting anomalous conditions with balanced sensitivity.

Although Oraculum presented consistent results, some competing approaches demonstrated advantages in specific scenarios. In the CityPulse dataset, Velrajan and Sharmila (2023) achieved the highest AA (95.4%), marginally above Oraculum's 94.6%. Their use of a particle swarm optimization strategy may offer benefits under traffic-intensive urban conditions, which are characteristic of the CityPulse dataset. Additionally, in the SmartSantander dataset, Liu et al. (2021) reported higher F1-scores in detecting anomalies under packet loss conditions, indicating that their unsupervised learning approach handled sensor communication noise effectively in that context.

Energy efficiency results also reflected architectural trade-offs. While Oraculum maintained competitive performance, Velrajan and Sharmila (2023) reported slightly higher efficiency in the CityPulse testbed. This behavior may stem from their lower action frequency, which reduces computational overhead, albeit at the cost of slower reactivity, an approach that may be suitable in energy-constrained IoT environments.

These observations underscored the influence of context in the design and evaluation of self-adaptive systems. Oraculum adopted a general-purpose model with broad support for metric types and rapid decision-making, whereas architectures such as those proposed by Tam et al. (2022) and Liu et al. (2021) exhibited performance gains under specific operating conditions. This evidence pointed to the potential of hybrid models that dynamically integrate or switch between strategies based on the environment.

[5] https://github.com/DarlanNoetzold/Oraculum-Model

Table 17
Performance of self-adaptive architectures on public benchmarks (mean values across 30 runs).

Model	MAT (s)	AA (%)	AO (%)	Stability (%)	RMSE	F1-score	Latency Reduction (%)	Benchmark
Oraculum (Ours)	0.07	95.2	3.8	97.5	0.21	0.89	27.4	SmartSantander
Tam et al. (2022)	2.84	91.8	5.6	88.9	0.32	0.81	21.6	SmartSantander
Velrajan et al. (2023)	1.52	93.5	4.2	90.7	0.28	0.84	24.1	SmartSantander
Liu et al. (2021)	1.74	90.2	6.1	87.3	0.35	0.76	18.9	SmartSantander
Oraculum (Ours)	0.09	94.6	3.9	96.3	0.24	0.87	25.1	CityPulse
Tam et al. (2022)	2.91	90.5	5.9	86.8	0.34	0.79	20.4	CityPulse
Velrajan et al. (2023)	1.43	95.4	4.3	92.5	0.27	0.86	26.2	CityPulse
Liu et al. (2021)	1.69	89.6	6.5	85.4	0.36	0.75	19.2	CityPulse
Oraculum (Ours)	0.06	93.1	3.5	95.7	0.18	0.90	28.7	Intel Lab Data
Tam et al. (2022)	2.77	91.2	5.1	89.1	0.29	0.83	23.5	Intel Lab Data
Velrajan et al. (2023)	1.48	92.7	4.6	91.5	0.26	0.86	25.6	Intel Lab Data
Liu et al. (2021)	1.64	94.3	6.4	90.1	0.30	0.78	22.4	Intel Lab Data

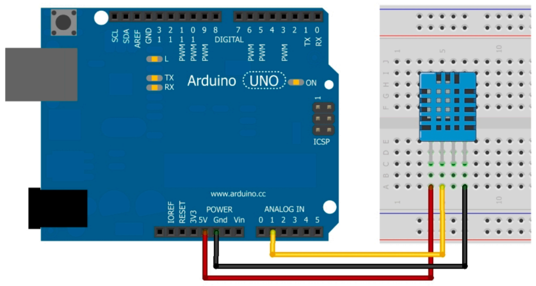

Fig. 14. Circuit schematic for connecting the DHT11 module to the Arduino Uno.

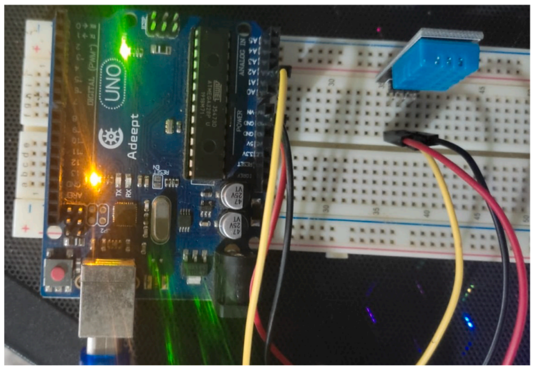

Fig. 15. Assembled circuit: Arduino Uno with DHT11 module.

Future work will include the development of meta-adaptation policies capable of leveraging reinforcement learning for selecting adaptation strategies according to workload patterns and system constraints. This direction may incorporate components from Liu et al. (2021) for anomaly classification and from Velrajan and Sharmila (2023) for energy-aware planning. Additional analysis of the internal AO per module will further clarify the cost-performance trade-offs.

5.5. Real-world experimental evaluation

Complementary tests using physical hardware components assess Oraculum in a more realistic context. The experimental setup approximates operational conditions by employing a real sensor, microcontroller, and distributed processing infrastructure. The main objective is to evaluate system adaptability and the data processing pipeline under real-world noise, hardware variability, and communication delays.

At the data capture layer, the system uses the DHT11 module for temperature and humidity acquisition. Unlike the bare sensor, the DHT11 module integrates signal conditioning and standardized connections, which simplifies wiring and increases robustness for prototyping and deployment. The module operates within a typical voltage range and provides reliable measurements of temperature and humidity, making it suitable for real-world IoT applications.

At the microcontroller interface, Fig. 14 presents the circuit schematic for connecting the DHT11 module to the Arduino Uno. The Arduino Uno is based on the ATmega328P microcontroller, featuring a 16 MHz clock, 2 KB SRAM, 32 KB flash memory, and 14 digital I/O pins. The module is powered directly from the Arduino's 5 V output, and data is transmitted via a single-wire protocol.

Fig. 15 shows the assembled circuit, where the Arduino Uno is programmed to collect temperature and humidity readings from the DHT11 module at one-second intervals. Each reading is transmitted to the processing layer for further analysis.

At the processing layer, the Raspberry Pi 4 Model B is used as the processing node. The Raspberry Pi 4 is equipped with a quad-core ARM Cortex-A72 CPU (1.5 GHz), 4 GB RAM, and runs the Raspbian operating system. Docker containers are used to replicate multiple processing nodes, enabling elastic allocation of CPU, RAM, and storage resources. The Raspberry Pi receives sensor data from the Arduino Uno via serial or network interface. Upon reception, the data undergoes pre-processing, including heuristic aggregation, filtering, and compression, following the same procedures adopted in the simulation environment. This layer is responsible for adapting the processing pipeline and can dynamically adjust resource allocation by scaling Docker containers.

At the core application layer, the pre-processed data is forwarded to a dedicated server running Ubuntu Server, equipped with an Intel Core i5 processor, 16 GB RAM, and a 1 TB SSD. This layer is responsible for storing the data, executing the Oraculum framework, and managing the adaptation of the processing infrastructure. The Oraculum framework can scale the number of processing containers, adjust resource allocation, and modify Apache server parameters (such as `MaxRequestWorkers`, `KeepAliveTimeout`, and thread pool size) to optimize load balancing and response time. Adaptation commands can also be sent back to the Raspberry Pi or Arduino Uno, enabling dynamic adjustment of the data collection interval or temporary suspension of data capture in response to system-level adaptation decisions.

A **virtualization layer** on the Arduino Uno enables scalability and adaptability testing without multiple physical devices. This approach uses a single DHT11 sensor to emulate multiple independent virtual sensors. For each acquisition cycle, the Arduino reads real temperature and humidity values, then generates N virtual sensor readings by applying small, deterministic offsets. Each virtual sensor receives a unique identifier (`id`) and its data is formatted as a separate JSON payload. This setup allows the processing and application layers to handle data streams as if they originated from N distinct sensors, supporting realistic testing of distributed scenarios, load, and adaptation policies.

Algorithm 2 Virtualized sensor data acquisition and transmission (Arduino Uno).

```
Require: DHT11 module connected, network interface configured,
    REST API endpoint on Raspberry Pi, default sampling interval $S$,
    number of virtual sensors $N$
1:  Initialize DHT11 module and network interface
2:  Set default values: $S = 60$ s, $N = 5$ virtual sensors
3:  Initialize JSON buffer and error counters
4:  while system is powered on do
5:    timestamp ← current system time
6:    for each virtual sensor $i$ from 1 to $N$ do
7:      Read humidity $h$ and temperature $t$ from DHT11
8:      if reading is valid then
9:        Apply deterministic offset: $h_i = h + (0.3 \times i)$, $t_i = t + (0.1 \times i)$
10:       Add noise variation: $h_i = h_i + \mathrm{random}(-0.1, 0.1)$, $t_i = t_i + \mathrm{random}(-0.05, 0.05)$
11:       Format JSON payload: {"id": $i$, "timestamp": timestamp, "humidity": $h_i$, "temperature": $t_i$}
12:       Attempt network connection to API endpoint
13:       if connection successful then
14:         Send HTTP POST request with JSON payload
15:         Wait for acknowledgment
16:         Close network connection
17:         Reset error counter for sensor $i$
18:       else
19:         Increment error counter for sensor $i$
20:         Store data in local buffer for retry
21:         Log network error for sensor $i$
22:       end if
23:     else
24:       Increment DHT11 error counter
25:       Log error: "Failed to read from DHT11 for virtual sensor $i$"
26:       if DHT11 error counter > 5 then
27:         Attempt DHT11 module reset
28:         Reset error counter
29:       end if
30:     end if
31:   end for
32:   if local buffer not empty then
33:     Attempt to flush buffered data to server
34:     if flush successful then
35:       Clear local buffer
36:     end if
37:   end if
38:   if adaptation command received via serial or network then
39:     Parse command type and parameters
40:     if command type = "SET_INTERVAL" then
41:       Update sampling interval: $S$ ← new value
42:     else if command type = "SET_SENSORS" then
43:       Update number of virtual sensors: $N$ ← new value
44:     else if command type = "RESET" then
45:       Reset all counters and buffers
46:     end if
47:     Send acknowledgment to processing layer
48:   end if
49:   Wait $S$ s before next acquisition cycle
50: end while
```

The Arduino is also capable of receiving adaptation commands from the processing layer, such as changes to the sampling interval or the number of virtual sensors, enabling dynamic reconfiguration during operation. The virtualization process includes error handling for individual virtual sensors, network retry mechanisms, and buffering capabilities to ensure data integrity. Algorithm 2 formalizes the complete workflow for virtualized data acquisition and transmission, including initialization, error handling, and adaptation command processing.

The offsets $\delta_h(i) = 0.3 \times i$ and $\delta_t(i) = 0.1 \times i$ ensure that each virtual sensor produces a unique baseline, while the added random noise simulates realistic sensor variations. This method enables the system to be stress-tested and evaluated as if it were operating with a much larger and more complex sensor network, while using only a single physical device. The buffering mechanism ensures that data is not lost during temporary network failures, and the adaptation command interface allows the system to dynamically reconfigure its behavior in response to changing conditions.

Algorithm 3 details the comprehensive workflow at the processing layer. The Raspberry Pi 4 acts as an intermediary processing node, receiving data from multiple virtual sensors, applying various data processing techniques, and managing resource allocation dynamically. Upon receiving sensor data, the system performs data validation, aggregation over configurable time windows, statistical filtering to remove outliers, and data compression to optimize network usage. All processing operations are containerized using Docker, enabling elastic scaling based on workload demands. The Raspberry Pi also serves as a bidirectional communication bridge, forwarding adaptation commands from

Algorithm 3 Data Pre-processing and Adaptation (Raspberry Pi 4)

```
Require: Incoming sensor data from Arduino Uno, Docker environment configured,
    connection to core server established, processing window size $W$, outlier
    threshold $\theta$
1:  Initialize data buffer, processing containers, and monitoring metrics
2:  Set default values: $W = 10$ samples, $\theta = 2.0$ standard deviations
3:  while system is active do
4:    while receiving data from Arduino do
5:      Receive JSON payload with sensor data (id, timestamp, humidity, temperature)
6:      Validate data format and ranges
7:      if data is valid then
8:        Add data to sensor-specific buffer for sensor id
9:        Update data reception metrics (count, rate, latency)
10:     else
11:       Log data validation error
12:       Increment error counter for sensor id
13:     end if
14:   end while
15:   for each sensor buffer with $\geq W$ samples do
16:     Extract window of $W$ most recent samples
17:     Calculate aggregation metrics: mean, min, max, standard deviation
18:     Apply outlier detection using $\theta$ threshold
19:     Remove outliers and recalculate statistics
20:     Apply smoothing filter (e.g., moving average)
21:     Extract features: trend, variance, anomaly score
22:     Compress processed data using gzip or similar
23:     Create processed payload with metadata
24:     Forward processed data to core application server
25:     Update processing metrics (throughput, latency, CPU usage)
26:   end for
27:   if adaptation command received from core server then
28:     Parse command type and parameters
29:     if command type = "SCALE_CONTAINERS" then
30:       Adjust Docker container resources (CPU limits, memory allocation)
31:       Scale number of processing containers up or down
32:     else if command type = "UPDATE_PARAMETERS" then
33:       Update processing parameters: $W$, $\theta$, compression level
34:     else if command type = "FORWARD_TO_ARDUINO" then
35:       Forward command to Arduino (sampling rate, sensor count, etc.)
36:       Wait for Arduino acknowledgment
37:     end if
38:     Send adaptation acknowledgment to core server
39:     Log adaptation action and resource changes
40:   end if
41:   if resource monitoring indicates overload then
42:     Request additional container resources from core server
43:     Temporarily increase buffer sizes
44:   else if resource utilization is low then
45:     Suggest resource reduction to core server
46:     Optimize container allocation
47:   end if
48:   Update system health metrics and send to core server
49: end while
```

the core server to the Arduino and reporting processing metrics back to the application layer.

The processing layer implements a multi-stage pipeline that includes data buffering, quality assessment, feature extraction, and adaptive resource management. Incoming data from each virtual sensor is first stored in a dedicated buffer, allowing for independent and parallel processing streams. Quality assessment routines validate data format and range, while feature extraction modules compute statistical summaries and detect trends or anomalies in the sensor readings. Adaptive resource management dynamically allocates CPU and memory to processing containers based on current workload and system health. Error handling mechanisms, such as retry logic and local buffering, ensure data integrity even in the presence of transient failures or network interruptions. Real-time monitoring capabilities continuously track processing performance, resource utilization, and system health, enabling prompt detection of bottlenecks or abnormal behavior.

The processing layer maintains separate buffers for each virtual sensor, enabling independent processing pipelines while sharing computational resources efficiently. The adaptive window size W and outlier threshold θ can be dynamically adjusted based on data characteristics, such as variance or detected anomalies, and system performance requirements. This flexible architecture ensures that the processing layer can handle varying data loads, maintain high data quality, and respond rapidly to adaptation commands, while providing detailed monitoring and feedback to the core application layer for coordinated system optimization.

Algorithm 4 describes the process at the core application server. Upon receiving pre-processed data from the processing layer, the server stores all incoming records in a database and continuously analyzes them using the Oraculum framework. This analysis includes anomaly detection, trend analysis, and performance monitoring. When an anomaly or suboptimal performance is detected, the system triggers adaptation actions, such as scaling the number of processing containers on the Raspberry Pi, adjusting resource limits (CPU, RAM, storage), and tuning Apache server parameters (e.g., `MaxRequestWorkers`, `KeepAliveTimeout`) to ensure optimal load balancing and responsiveness. The core server can also send adaptation commands downstream, instructing the processing layer or Arduino to adjust the data capture rate, change the number of virtual sensors, or temporarily pause data collection to prevent overload or data loss.

Algorithm 4 Core application and system adaptation (Server).

```
Require: Pre-processed data from Raspberry Pi, Oraculum framework,
    Docker and Apache configuration access
1: while receiving data do
2:     Store incoming data in database
3:     Analyze metrics for anomalies and performance trends
4:     if adaptation required then
5:         Scale Docker containers on Raspberry Pi (increase/decrease resources)
6:         Adjust Apache server parameters (e.g., MaxRequestWorkers,
           KeepAliveTimeout)
7:         Send adaptation commands to Raspberry Pi and Arduino (e.g.,
           adjust sampling rate)
8:     end if
9: end while
```

This methodology enables the evaluation of the Oraculum framework in a real-world scenario, with end-to-end adaptivity from sensor data capture to distributed processing and core application adaptation. The use of the DHT11 module, as opposed to the bare sensor, increases reliability and ease of integration, which is particularly relevant for practical deployments. All figures referenced (DHT11 module, circuit schematic, assembled circuit, and Raspberry Pi 4) illustrate the experimental setup and hardware components used in the tests.

Figs. 16–18 present the main results of the real-world experiments, which were conducted over a 24-h period, totaling 1440 iterations (one iteration every 60 s). This setup was chosen to simulate a realistic, long-term deployment scenario, allowing the system to experience natural environmental variations, operational fluctuations, and real-world network conditions.

At the data capture layer (Fig. 16), the Arduino Uno with the DHT11 module consistently collected and transmitted temperature and humidity data. The readings reflect expected daily environmental changes, with some minor fluctuations corresponding to natural variations in the test environment. The system maintained stable operation for most of the experiment, with no significant data loss or sensor failures.

The processing layer (Fig. 17) shows the number of received sensor readings, pre-processing latency, and resource utilization on the Raspberry Pi 4. For the majority of the 24-h period, the system operated within expected resource bounds, with CPU and memory usage remaining stable. Notably, at iteration 750, a real network outage occurred, which is visible as a gap in the data stream. During this event, the system automatically buffered incoming data and reduced resource allocation, demonstrating its ability to adapt to connectivity issues without losing information.

At the core application layer (Fig. 18), the server monitored data ingestion rates and triggered adaptation actions in response to detected anomalies or performance changes. The network outage at iteration 750 led to a temporary drop in data rate and the activation of several adaptation mechanisms, including scaling down Docker containers, pausing non-essential tasks, and buffering data. After network restoration at iteration 820, the system gradually resumed normal operation, restored resource allocation, and stabilized throughput.

Over the 24-h, 1,440-iteration experiment, the Oraculum system achieved a MAT of 0.10 s, demonstrating that the system was able to detect changes and execute adaptation actions almost immediately after identifying an event or anomaly. The AA reached 92.4%, indicating that the majority of adaptation decisions were appropriate and effective, with a low incidence of false triggers or missed adaptations. The AO was 5.6%, reflecting the additional computational and resource cost introduced by the adaptation mechanisms relative to baseline operation; this low value highlights the efficiency of the system's adaptive logic. System stability was measured at 95.1%, meaning that the system maintained its intended operational state for nearly the entire experiment, with only brief deviations during significant events such as the network outage. Additionally, the average latency reduction was 22.1%, showing that the adaptive strategies not only preserved system responsiveness but also improved it compared to a static configuration. Collectively, these results confirm that the Oraculum system maintained high performance, reliability, and adaptability throughout the real-world experiment, even in the presence of environmental fluctuations and unexpected disruptions.

Throughout the experiment, the system demonstrated robust self-adaptation capabilities. During periods of environmental fluctuation (e.g., temperature and humidity spikes), the system dynamically adjusted anomaly detection thresholds and sampling intervals, reducing false positives and optimizing network usage. When processing load increased, the system allocated additional Docker containers to maintain low latency, and scaled down resources when the load decreased, optimizing energy and resource consumption.

The most significant event was the real network outage at iteration 750. The system detected the failure within 2 min, scaled down processing resources, paused non-essential tasks, and buffered up to 70 records locally. This ensured that no data was lost and that the system could recover gracefully. Upon network restoration at iteration 820, all tasks were resumed, resources were gradually reallocated, and the system returned to stable operation within a few minutes. This event highlights

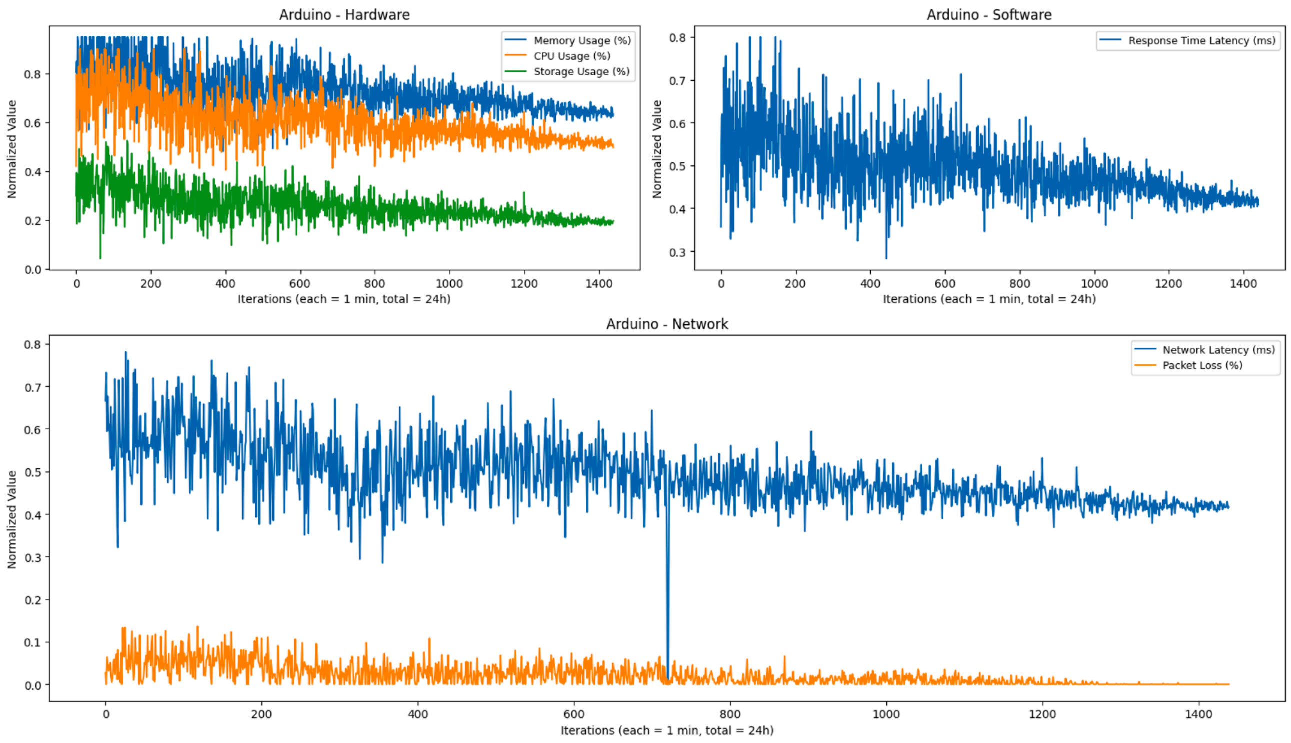

Fig. 16. Data capture layer: temperature and humidity readings (Arduino Uno with DHT11 module).

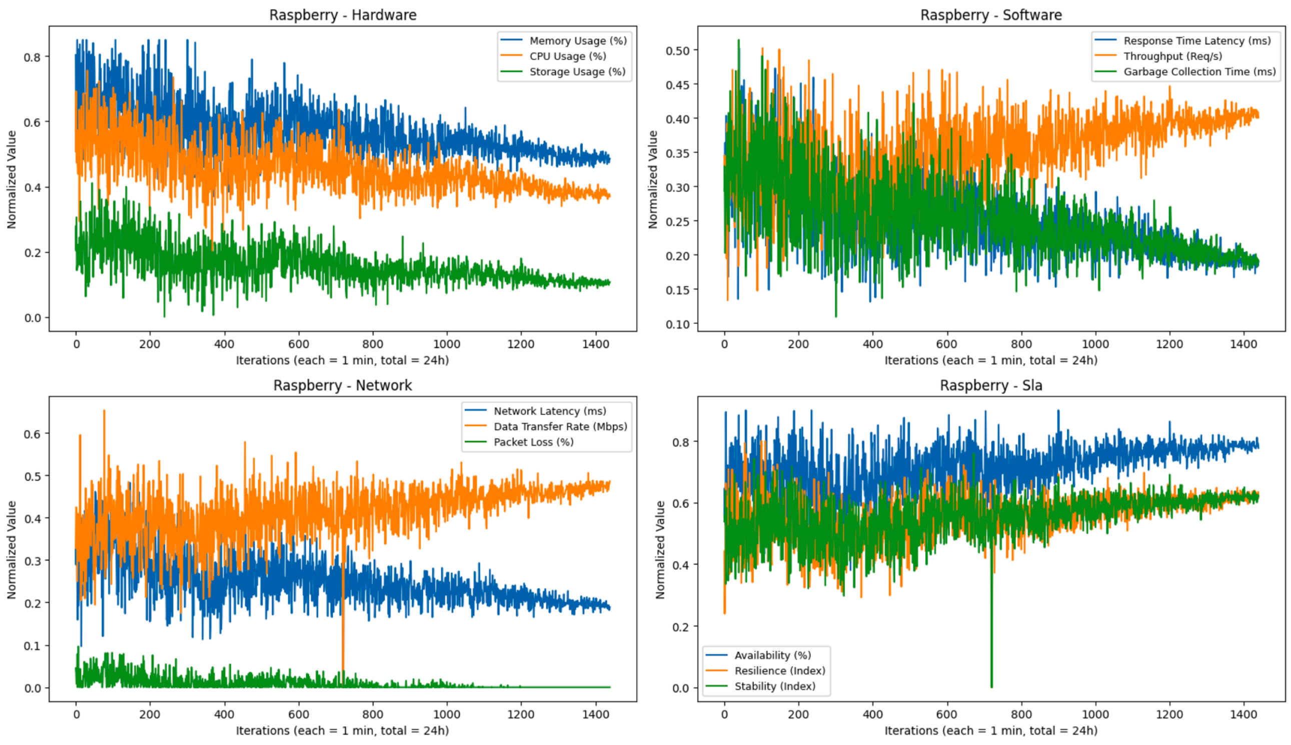

Fig. 17. Processing layer: received data and resource usage (Raspberry Pi 4).

the framework's ability to maintain data integrity and service continuity even in the face of real disruptions.

During prolonged periods of stability (iterations 900-1,300), the frequency of adaptation actions decreased significantly, indicating that the system was able to maintain optimal performance with minimal intervention. Minor events, such as micro-failures or brief environmental changes, were handled automatically and did not impact overall system performance.

Table 18 summarizes the main adaptation actions and their outcomes throughout the 1,440-iteration, 24-h experiment. The diversity and effectiveness of these actions demonstrate the system's capacity for both proactive and reactive adaptation in a real-world context. During system

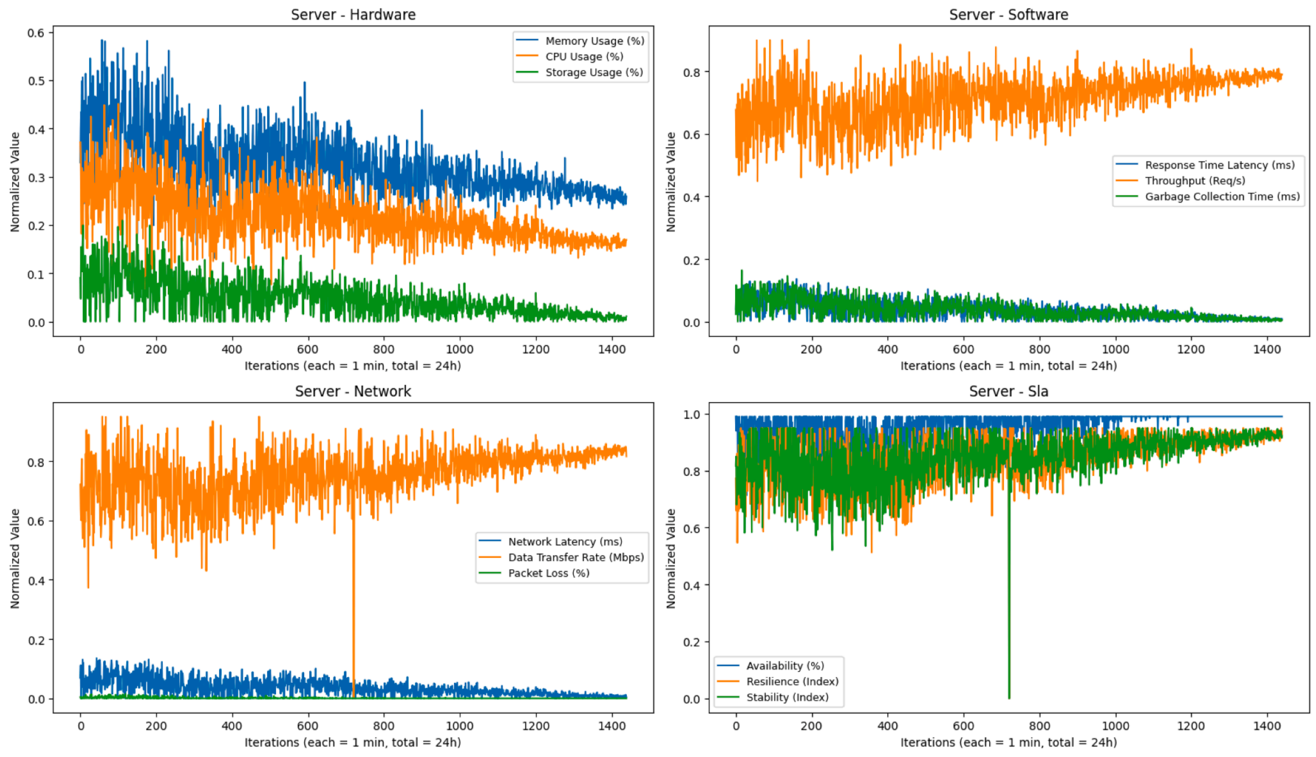

Fig. 18. application layer: data ingestion and adaptation actions (server).

initialization, actions such as time synchronization, anomaly threshold calibration, and network handshake retries ensured that all components started in a consistent and reliable state. Early in the experiment, the system performed Docker image updates and log rotations to maintain up-to-date and efficient operation.

Throughout the stable operation phase, the system responded to environmental fluctuations, such as temperature and humidity spikes, by dynamically updating anomaly detection thresholds and activating outlier filters, which reduced false positives and improved the accuracy of anomaly detection. The sampling interval was adjusted in response to changing conditions, optimizing network and sensor resource usage. The system also performed periodic sensor calibration checks, data aggregation window adjustments, and data validation routines to maintain data quality and integrity.

Resource management actions were frequent, including scaling Docker containers up or down in response to processing load, performing memory optimizations, and releasing idle resources to ensure efficient operation. The system also enabled data compression and log rotation to reduce network and disk usage, and performed regular data backups to safeguard against data loss.

During the network outage event, the system quickly detected the failure, scaled down containers, paused non-essential tasks, expanded the local buffer, and activated power-saving modes to preserve resources. Adaptive sampling was used to further reduce sensor workload during the outage. Health checks and data integrity verifications ensured that all modules remained operational and that buffered data was not corrupted. Upon network restoration, the system resumed paused tasks, flushed the buffered data, reallocated resources, and synchronized logs, enabling a smooth and rapid recovery.

In the later stages of the experiment, as the environment stabilized, the system gradually deactivated outlier filters, reset aggregation windows, and scaled down resources to minimize energy consumption. Final data validation, log synchronization, and sensor self-tests were performed to confirm the integrity and completeness of the experiment. Throughout the final stabilization period, minimal intervention was required, reflecting the system's ability to maintain optimal performance autonomously.

These actions illustrate the system's comprehensive approach to adaptation, encompassing environmental monitoring, resource management, fault tolerance, and data integrity, and highlight its readiness for deployment in dynamic, real-world smart environments. The results confirm that Oraculum is capable of maintaining high performance, reliability, and adaptability in real-world deployments, even when faced with unexpected events such as network outages. The system's ability to automatically adjust its operation, recover from failures, and minimize resource usage during stable periods is essential for practical, long-term smart environment applications.

6. Limitations and discussion

Despite the structured architecture and consistent evaluation of the Oraculum model, several limitations remain. The model's generalization capability may be limited due to potential overfitting to specific workload patterns, highlighting the need for further validation in heterogeneous environments. Sensitivity to non-stationary data in dynamic smart environments can lead to delayed adaptation, suggesting that concept drift detection and adaptive retraining strategies should be explored.

The semantic processing layer, which relies on manually defined ontologies, constrains scalability and transferability across domains. Future work should investigate ontology alignment and automatic concept extraction to address this limitation. Failure injection experiments showed that while the system recovers quickly from isolated failures, cascading failures can increase recovery time, indicating that decentralized or multi-agent adaptation mechanisms may be beneficial. Hyperparameter choices, such as the discount factor and learning rate, significantly affect adaptation speed and stability, and adaptive scheduling of these parameters could improve performance. The use of reinforcement learning algorithms like TD3 introduces computational overhead and requires careful tuning of the exploration-exploitation balance.

Table 18
Main adaptation actions and outcomes during the 1,440-iteration experiment.

Event/Iteration	Action	Quantity	Qualitative Outcome	Quantitative Result
0–100	System initialization and self-test	1 cycle	Sensor, network, and container check	All services active, initial latency 0.12 s
10	Time sync adjustment	1 update	RTC synchronized with NTP server	Timestamp drift <1s
25	Initial anomaly threshold calibration	1 calibration	Baseline for anomaly detection set	Initial F1-score 0.78
50	Network handshake retry	2 attempts	Ensured stable connection	No packet loss
75	Docker image update	1 update	Pulled latest container image	No downtime
101–700	Stable operation	–	Continuous data flow, minor environmental fluctuations	Avg. latency 0.09 s, CPU usage 35–45%
120	Temperature spike	1 threshold update	Anomaly detection threshold increased	6% reduction in false positives
135	Humidity spike	1 threshold update	Anomaly detection threshold increased	5% reduction in false positives
150	Outlier filtering	1 filter activation	Outlier filter enabled for humidity	RMSE reduced by 0.02
180	Sampling interval adjustment	1 change	Sampling interval set to 2 min for 20 min	Network usage reduced by 10%
200	Sensor micro-failure	1 restart	Automatic sensor driver restart	Recovery in 2 min, no data loss
220	Data aggregation window change	1 update	Aggregation window set to 10 samples	Smoother trend, less noise
250	CPU usage spike	1 scaling up	+ 1 Docker container allocated	Latency reduced from 0.14 s to 0.10 s
275	Memory usage optimization	1 garbage collection	Cleared unused memory in containers	Memory usage reduced by 8%
300	Load decrease	1 scaling down	−1 Docker container removed	CPU usage reduced by 10%
320	Log rotation	1 operation	Rotated and compressed logs	Disk usage reduced by 5%
350	Data compression enabled	1 activation	Enabled gzip compression for payloads	Network usage reduced by 7%
400	Environmental fluctuation	1 dynamic threshold	Adaptive threshold for temperature	F1-score improved to 0.81
420	Micro network delay	1 retry	Automatic resend of failed packet	No data loss
450	Sensor calibration check	1 check	Verified sensor accuracy	No recalibration needed
500	Data backup	1 backup	Backed up local buffer to SD card	Data redundancy ensured
520	Network latency spike	1 alert	Alerted core server, increased timeout	No dropped connections
600	Processing load peak	1 scaling up	+ 1 Docker container allocated	Throughput stabilized
650	Idle resource release	1 operation	Released unused system resources	CPU usage reduced by 5%
700	Load decrease	1 scaling down	−1 Docker container removed	Energy consumption reduced
720	Data validation routine	1 check	Validated data integrity	No errors found
750 (Network outage)	Failure detection	1 event	Automatic network loss detection	Detection time: 2 min
750 (Network outage)	Scale down containers	2 → 1 container	Resource reduction for contingency mode	RAM usage reduced by 35%
750 (Network outage)	Pause non-essential tasks	2 tasks paused	Prevented overload and data loss	No data discarded
755	Local buffer expansion	1 operation	Increased buffer size to 100 records	No data overflow
760	Adaptive sampling reduction	1 change	Sampling interval set to 3 min during outage	Reduced sensor workload
770	Health check routine	1 check	Ensured all modules responsive	All modules healthy
780	Power-saving mode activation	1 mode change	Reduced peripheral power usage	Power draw reduced by 12%
800	Data integrity check	1 verification	Verified buffered data	No corruption detected
810	Buffer flush attempt	1 attempt	Tried to send buffered data	Awaiting network restoration
820 (Restoration)	Resume tasks	2 tasks resumed	Gradual system recovery	Latency spike (0.21 s), normalized in 3 min
825	Buffer flush	1 operation	Sent all buffered data to server	Data backlog cleared
830	Resource reallocation	1 scaling up	1 → 2 containers	Processing capacity restored
850	Log sync	1 operation	Synced local logs with server	Logs up-to-date
900	Environmental variation	1 threshold update	Dynamic anomaly threshold update	F1-score kept above 0.80
950	Micro network failure	1 reconnection	Automatic reconnection	Loss of 1 sample, no impact on flow
1,000	Data aggregation window reset	1 update	Aggregation window set back to 5 samples	More responsive trend
1,050	Outlier filter deactivation	1 change	Disabled outlier filter after stable period	Slight increase in RMSE
1,100	Log rotation	1 operation	Rotated and compressed logs	Disk usage reduced by 3%
1,150	Power-saving mode deactivation	1 mode change	Restored normal power usage	Full performance restored
1,200	Sensor self-test	1 test	Verified sensor health	No issues found
1,250	Data backup	1 backup	Backed up buffer to SD card	Data redundancy ensured
1,300	Final scaling down	1 operation	−1 Docker container removed	Minimal resource usage
1,350	Final data validation	1 check	Validated all data for errors	No errors found
1,400	Final log sync	1 operation	Synced all logs to server	Logs complete
1,401–1,440	Final stabilization	–	Minimal intervention required	No adaptation actions registered

Current evaluation is based on a simulated environment, which, despite its realism, cannot capture all complexities of real-world deployments, such as unpredictable network failures or hardware heterogeneity. Security-related metrics are not yet included, limiting applicability in adversarial scenarios. Future work will focus on deployment in physical testbeds, integration of security-aware adaptation, and more robust statistical reporting. These limitations do not undermine the utility of Oraculum but highlight areas for further refinement and enhancement.

7. Conclusion

This work introduced Oraculum, a self-adaptive model designed to optimize real-time data processing, predictive analytics, and automated decision-making in intelligent environments. By integrating continuous metric monitoring, anomaly forecasting, and a reinforcement learning (RL)-based adaptation loop, Oraculum dynamically reconfigures system behavior in response to contextual changes, addressing the challenges of heterogeneous and dynamic operational settings.

The experimental evaluation was conducted in two complementary stages: first, using the SHiELD simulator to enable large-scale, repeatable experiments under controlled conditions; and second, deploying Oraculum in a real-world architecture with physical devices, including an Arduino Uno with a DHT11 sensor and a Raspberry Pi 4. In the simulated environment, Oraculum demonstrated its ability to effectively monitor and respond to performance variations across multiple system layers, with predictive models accurately anticipating potential degradations and enabling proactive adaptations. Among the RL algorithms tested, the TD3 agent achieved the best balance between adaptation latency, system stability, and overhead control.

In the real-world deployment, the system maintained robust performance over 24 h and 1440 acquisition cycles, even in the presence of environmental fluctuations and real network disruptions. The MAT was 0.10 s, AA reached 92.4%, AO remained at 5.6%, and operational stability was 95.1%. These results confirm that Oraculum's adaptation mechanisms are effective not only in simulated scenarios but also in practical, heterogeneous environments, with only a modest reduction in performance due to real-world uncertainties.

Compared to recent state-of-the-art approaches, such as deep Q-network (DQN)-based resource allocation in cloud-edge environments (Cen & Li, 2022) and Kubernetes-extended self-healing mechanisms for IoT infrastructure (Samarakoon et al., 2023), Oraculum demonstrated demonstrated highly competitive performance in terms of response time and resource overhead, without compromising accuracy or stability. These results substantiate the model's contributions by showing that the combination of predictive analytics with RL-driven closed-loop adaptation can significantly enhance system responsiveness and resource efficiency in complex, real-time environments.

The main contributions of this work lie in the integration of predictive modeling with reinforcement learning to create a closed-loop adaptation system that is both scalable and efficient. The results presented clearly support these contributions by demonstrating measurable improvements in adaptation speed, accuracy, and operational stability across diverse system layers. This establishes Oraculum as a practical and effective solution for dynamic resource management in intelligent environments, advancing the state of the art in self-adaptive systems.

The analysis of resource usage indicated consistent trends: CPU and memory consumption initially increased during adaptation phases but stabilized afterward. Network-level indicators, such as latency and packet loss, improved following adaptive actions. Although variability in security-related metrics was observed, the model demonstrated potential for further integration of adaptive security policies (Zhang et al., 2025b).

The current evaluation did not explicitly cover security-related metrics such as exposure to adversarial attacks, data poisoning, or maliciously crafted input patterns. The experiments focused on performance and reliability under non-malicious conditions, and Oraculum's stability under coordinated attacks or abnormal adversarial inputs therefore remains outside the validated scope. As a result, the present version of Oraculum should not be interpreted as a fully hardened solution against security threats, and its behavior under active attacks constitutes a limitation of this study.

Despite the promising outcomes, several limitations require further attention. The validation process with the simulator enabled large-scale and repeatable testing, while the real-world experiments confirmed the feasibility and robustness of the approach in a physical architecture. However, both stages were limited in scale: the real-world deployment involved a modest number of devices, and the simulator, while flexible, cannot fully capture the diversity and unpredictability of large, heterogeneous hardware infrastructures.

The evaluation also did not include explicit security-related scenarios such as adversarial attacks or data poisoning campaigns, which limited the analysis of Oraculum's behavior under malicious interference. More comprehensive tests with larger and more complex hardware architectures, including controlled security stress tests, remain as future work. Future research directions include:

- Deploying and validating Oraculum in larger-scale, real-world intelligent environments to assess its behavior under uncontrolled and noisy conditions.
- Expanding the monitored metrics to include energy consumption, power efficiency, and security-related indicators (e.g., intrusion attempts, anomaly score distributions under suspected attacks, and resilience to data poisoning), broadening the model's scope of adaptation.
- Adopting distributed RL techniques to enable cooperative adaptation across decentralized infrastructures.
- Designing and validating an adaptive security layer capable of reacting to adversarial inputs and evolving threat patterns through policy-based RL agents, including experiments with adversarial and poisoned data streams.
- Analyzing the long-term impact of continuous adaptation cycles on overall system reliability and maintenance requirements.

These future directions aim to strengthen Oraculum's applicability to a broader range of scenarios, contributing to the development of intelligent systems capable of autonomously optimizing their performance in dynamic and resource-constrained environments.

CRediT authorship contribution statement

Darlan Noetzold: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Resources, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition; **Valderi Reis Quietinho Leithardt:** Conceptualization, Methodology, Validation, Formal analysis, Supervision, Writing – review & editing, Funding acquisition; **Juan Francisco de Paz Santana:** Conceptualization, Methodology, Resources, Validation, Supervision, Writing – review & editing, Funding acquisition; **Jorge Luis Victória Barbosa:** Conceptualization, Methodology, Validation, Supervision, Project administration, Writing – review & editing, Funding acquisition.

Data availability

Data will be made available on request.

Declaration of competing interests

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

The authors acknowledge financial support from CNPq (National Council for Scientific and Technological Development, grant 307137/2022-8) and CAPES (Coordination for the Improvement of Higher Education Personnel - Brazil, Finance Code 001). The authors also thank Unisinos (University of Vale do Rio dos Sinos), ISCTE-IUL (University Institute of Lisbon), and the University of Salamanca for their institutional support throughout this research. This work was partially funded by national funds through FCT - Foundation for Science and Technology, I.P., under projects UIDB/04466/2025, UIDP/04466/2025, and project 16,881 (LISBOA2030-FEDER-00816400). Additional support was provided by the Consejería de Educación de la Junta de Castilla y León through the research group ESAL - Expert System and Applications Lab (ESALAB).

References

Aguayo, O., Sepúlveda, S., & Mazo, R. (2024). Variability management in self-adaptive systems through deep learning: A dynamic software product line approach. *Electronics*, *13*(5), 905. https://doi.org/10.3390/electronics13050905

Al-Sharafi, M. A., Iranmanesh, M., Al-Emran, M., Alzahrani, A. I., Herzallah, F., & Jamil, N. (2023). Determinants of cloud computing integration and its impact on sustainable performance in SMEs: An empirical investigation using the SEM-ANN approach. *Heliyon*, *9*(5), e16299. https://doi.org/10.1016/j.heliyon.2023.e16299

Alfonso, I., Garcés, K., Castro, H., & Cabot, J. (2021). Self-adaptive architectures in iot systems: A systematic literature review. *Journal of Internet Services and Applications*, *12*(1), 14. https://doi.org/10.1186/s13174-021-00145-8

Arias del Campo, F., Guevara Neri, M. C., Vergara Villegas, O. O., Cruz Sánchez, V. G., de Jesús Ochoa Domínguez, H., & García Jiménez, V. (2021). Auto-adaptive multilayer perceptron for univariate time series classification. *Expert Systems with Applications*, *181*, 115147. https://doi.org/10.1016/j.eswa.2021.115147

Aydemir, O. (2021). A new performance evaluation metric for classifiers: polygon area metric. *Journal of Classification*, *38*, 1–23. https://doi.org/10.1007/s00357-020-09362-5

Barrett, E., Howley, E., & Duggan, J. (2013). Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, *25*(12), 1656–1674. https://doi.org/10.1002/cpe.2864

Caporuscio, M., D'Angelo, M., Grassi, V., & Mirandola, R. (2016). Reinforcement learning techniques for decentralized self-adaptive service assembly. In M. Aiello, E. B. Johnsen, S. Dustdar, & I. Georgievski (Eds.), *Service-oriented and cloud computing* (pp. 53–68). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-44482-6_4

Cen, J., & Li, Y. (2022). Resource allocation strategy using deep reinforcement learning in cloud-edge collaborative computing environment. *Security and Communication Networks*, *2022*, Article ID 9597429. https://doi.org/10.1155/2022/9597429

Dehghan Shoorkand, H., Nourelfath, M., & Hajji, A. (2024). A hybrid CNN-LSTM model for joint optimization of production and imperfect predictive maintenance planning. *Reliability Engineering & System Safety*, *241*, 109707. https://doi.org/10.1016/j.ress.2023.109707

Ding, X., An, Z., Rathee, A., & Du, W. (2025). A safe and data-efficient model-based reinforcement learning system for HVAC control. *IEEE Internet of Things Journal*, *12*(7), 8014–8032. https://doi.org/10.1109/JIOT.2025.3540402

Emmanuel Sapnken, F., & Gaston Tamba, J. (2022). Petroleum products consumption forecasting based on a new structural auto-adaptive intelligent grey prediction model. *Expert Systems with Applications*, *203*, 117579. https://doi.org/10.1016/j.eswa.2022.117579

Etemadi, M., Ghobaei-Arani, M., & Shahidinejad, A. (2021). A cost-efficient auto-scaling mechanism for iot applications in fog computing environment: A deep learning-based approach. *Cluster Computing*, *24*, 3277–3292. https://doi.org/10.1007/s10586-021-03307-2

Fang, X., Chen, Y., Bhuiyan, Z. A., He, X., Bian, G., Crespi, N., & Jing, X. (2025). Mixer-transformer: Adaptive anomaly detection with multivariate time series. *Journal of Network and Computer Applications*, *241*, 104216. https://doi.org/10.1016/j.jnca.2025.104216

Gupta, A., & Agarwal, S. (2025). NCDT-CSS: Enhancing performance using noncoherent distributed transmission of chirp spread spectrum. *IEEE Internet of Things Journal*, *12*(7), 7969–7979. https://doi.org/10.1109/JIOT.2025.3532298

Hameed, A., Violos, J., Santi, N., Leivadeas, A., & Mitton, N. (2021). A machine learning regression approach for throughput estimation in an IoT environment, . (pp. 29–36). https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics53846.2021.00020

Jamshidi, S., Amirnia, A., Nikanjam, A., Nafi, K. W., Khomh, F., & Keivanpour, S. (2025). Self-adaptive cyber defense for sustainable IoT: A DRL-based IDS optimizing security and energy efficiency. *Journal of Network and Computer Applications*, *239*, 104176. https://doi.org/10.1016/j.jnca.2025.104176

Jaskierny, L., & Kotulski, L. (2023). A self-adapting IoT network configuration supported by distributed graph transformations. *Appl. Sci.*, *13*(23), 12718. https://doi.org/10.3390/app132312718

Lab, I. B. R. (2004). Intel lab data dataset. http://db.csail.mit.edu/labdata/labdata.html. Accessed on: Jul 11, 2025.

Liu, D., Zhen, H., Kong, D., Chen, X., Lei, Z., Yuan, M., & Wang, H. (2021). Sensors anomaly detection of industrial internet of things based on isolated forest algorithm and data compression. *Scientific Programming*, *2021*, 1–9. https://doi.org/10.1155/2021/6699313

Liu, Y., Liao, G., Jiang, G., Chen, Y., Cui, Y., Xu, H., & Yu, M. (2024). Multi-exposure fused light field image quality assessment for dynamic scenes: Benchmark dataset and objective metric. *Expert Systems with Applications*, *256*, 124881. https://doi.org/10.1016/j.eswa.2024.124881

Luo, J., Wang, Z., Shen, C., Kuijper, A., Wen, Z., & Liu, S. (2016). Modeling and implementation of multi-position non-continuous rotation gyroscope north finder. *Sensors*, *16*(9). https://www.mdpi.com/1424-8220/16/9/1513. https://doi.org/10.3390/s16091513

Martinez, J., Barber, R., & Civit, A. (2022). On managing knowledge for MAPE-k loops in self-adaptive robotic systems. *Applied Sciences*, *12*(17), 8583. https://doi.org/10.3390/app12178583

Morabito, R. (2017). Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, *5*, 8835–8850. https://doi.org/10.1109/ACCESS.2017.2704444

Padgham, L., & Winikoff, M. (2004). Developing intelligent agent systems: a practical guide. John Wiley & Sons Ltd. https://doi.org/10.1002/0470861223

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Proceedings of the 12th international conference on evaluation and assessment in software engineering* (pp. 68–77). Swindon, UK: BCS Learning & Development Ltd. https://doi.org/10.14236/ewic/EASE2008.8

Pinthurat, W., Surinkaew, T., & Hredzak, B. (2024). An overview of reinforcement learning-based approaches for smart home energy management systems with energy storages. *Renewable and Sustainable Energy Reviews*, *202*, 114648. https://doi.org/10.1016/j.rser.2024.114648

Priya, S. A., Bhat, N., Kanna, B. R., Rajalakshmi, S., Jeyavathana, R. B., & S, S. (2024). Proactive network optimization using deep learning in predicting iot traffic dynamics. In *2024 4th international conference on innovative practices in technology and management (ICIPTM)* (pp. 1–6). https://doi.org/10.1109/ICIPTM59628.2024.10563433

Raibulet, C., Oh, J., & Leest, J. (2023). Analysis of MAPE-k loop in self-adaptive systems for cloud, IoT and CPS. In *Service-oriented computing - ICSOC 2022 workshops* (pp. 130–141). Springer. https://doi.org/10.1007/978-3-031-26507-5_11

Restuccia, F., & Melodia, T. (2020). DeepWiERL: Bringing deep reinforcement learning to the internet of self-adaptive things. In *Proc. IEEE INFOCOM* (pp. 844–853). https://doi.org/10.1109/INFOCOM41043.2020.9155461

Sah, D. K., Nguyen, T. N., Cengiz, K., Dumba, B., & Kumar, V. (2022). Load-balance scheduling for intelligent sensors deployment in industrial internet of things. *Cluster Computing*, *25*(3), 1715–1727. https://doi.org/10.1007/s10586-021-03316-1. https://doi.org/10.1007/s10586-021-03316-1

Samarakoon, S., Bandara, S., Jayasanka, N., & Hettiarachchi, C. (2023). Self-healing and self-adaptive management for IoT-edge computing infrastructure. In *2023 moratuwa engineering research conference (MERCon)* (pp. 473–478). https://doi.org/10.1109/MERCon60487.2023.10355514

Sanchez, L., & Galache, J. (2024). SmartSantander: A real-world iot testbed for smart cities. https://www.smartsantander.eu. Accessed on: Jul 11, 2025.

SAP (2007). Standardized technical architecture modeling: Conceptual and design level. Available at: https://help.sap.com/docs/SAP_POWERDESIGNER, Accessed in: March 28, 2025.

Shi, J., Zhu, G., Li, H., Hawbani, A., Wu, J., Lin, N., & Zhao, L. (2025). Multi-AAVs flocking for navigation and obstacle avoidance in network-constrained environments. *IEEE Internet of Things Journal*, *12*(7), 8931–8946. https://doi.org/10.1109/JIOT.2024.3507782

Shin, M., Kim, M., Park, G., & Abraham, A. (2023). Adaptive variable sampling model for performance analysis in high cache-performance computing environments. *Heliyon*, *9*(6), e16777. https://doi.org/10.1016/j.heliyon.2023.e16777

Sobieraj, M., & Kotyński, D. (2024). Docker performance evaluation across operating systems. *Applied Sciences*, *14*(15), 6672. https://doi.org/10.3390/app14156672

Steventon, A., & Wright, S. (2006). Intelligent spaces: the application of pervasive ICT. London, UK: Springer-Verlag. Accessed on: Jul 11, 2025. https://doi.org/10.1007/978-1-84628-429-8

Sun, W., Tohirovich Dedahanov, A., Li, W. P., & Young Shin, H. (2024). Sanctions and opportunities: Factors affecting china's high-tech SMEs adoption of artificial intelligence computing leasing business. *Heliyon*, *10*(16), e36620. https://doi.org/10.1016/j.heliyon.2024.e36620

Tam, P., Math, S., & Kim, S. (2022). Priority-aware resource management for adaptive service function chaining in real-time intelligent IoT services. *Electronics*, *11*(19), 2976. https://doi.org/10.3390/electronics11192976

Tang, K., Chen, S., Guo, T., Ma, Y., & Khattak, A. J. (2024). An adaptive deep multi-task learning approach for citywide travel time collaborative estimation. *Expert Systems with Applications*, *244*, 123009. https://doi.org/10.1016/j.eswa.2023.123009

Velrajan, S., & Sharmila, V. C. (2023). QoS-aware service migration in multi-access edge computing using closed-loop adaptive particle swarm optimization algorithm. *Journal of Network and Systems Management*, *31*(1), 17. https://doi.org/10.1007/s10922-022-09707-y

Wang, Q., Su, F., Dai, S., Lu, X., & Liu, Y. (2024). AdaGC: A novel adaptive optimization algorithm with gradient bias correction. *Expert Systems with Applications*, *256*, 124956. https://doi.org/10.1016/j.eswa.2024.124956

Wang, X., Luo, Q., Liu, K., Mao, R., & Wu, G. (2025). Deep learning method based on multiscale enhanced feature fusion for vehicle behavior prediction. *IEEE Internet of Things Journal*, *12*(7), 9142–9155. https://doi.org/10.1109/JIOT.2024.3508034

Xu, M., & Buyya, R. (2019). BrownoutCon: A software system based on brownout and containers for energy-efficient cloud computing. *Journal of Systems and Software*, *155*, 91–103. https://doi.org/10.1016/j.jss.2019.05.024

Yadav, A., & Vishwakarma, D. K. (2023). MRT-Net: Auto-adaptive weighting of manipulation residuals and texture clues for face manipulation detection. *Expert Systems with Applications*, *232*, 120898. https://doi.org/10.1016/j.eswa.2023.120898

Yang, F., Ge, S., Liu, J., Yan, K., Gao, A., Dong, Y., Yang, M., & Zhang, W. (2025). High-precision short-term industrial energy consumption forecasting via parallel-NN with adaptive universal decomposition. *Expert Systems with Applications*, *289*, 128366. https://doi.org/10.1016/j.eswa.2025.128366

Yang, Z., Abbasi, I. A., Mustafa, E. E., Ali, S., & Zhang, M. (2021). An anomaly detection algorithm selection service for IoT stream data based on tsfresh tool and genetic algorithm. *Security and Communication Networks*, *2021*(1), 6677027. https://doi.org/10.1155/2021/6677027

Zeshan, F., Ahmad, A., Babar, M. I., Hamid, M., Hajjej, F., & Ashraf, M. (2023). An IoT-enabled ontology-based intelligent healthcare framework for remote patient monitoring. *IEEE Access*, *11*, 133947–133966. https://doi.org/10.1109/ACCESS.2023.3332708

Zhang, J., Liu, G., Chen, J., & Cheng, Y. (2025a). Multi-scale adaptive residual cold diffusion model for low-dose CT denoising. *Expert Systems with Applications*, *294*, 128817. https://doi.org/10.1016/j.eswa.2025.128817

Zhang, X., Guo, H., Zhang, Z., Tang, G., Sun, J., Shen, Y., & Ma, J. (2025b). Effectively detecting software vulnerabilities via leveraging features on program slices. *IEEE Internet of Things Journal*, *12*(7), 8033–8048. https://doi.org/10.1109/JIOT.2025.3541090

Zhang, Y., Zhang, H., Yang, Y., Sun, W., Zhang, H., & Fu, Y. (2025c). Adaptive differential privacy in asynchronous federated learning for aerial-aided edge computing. *Journal of Network and Computer Applications*, *235*, 104087. https://doi.org/10.1016/j.jnca.2024.104087

Zhao, S., & Guo, M. (2024). Electric vehicle power system in intelligent manufacturing based on soft computing optimization. *Heliyon*, *10*(21), e38946. https://doi.org/10.1016/j.heliyon.2024.e38946