UNISINOS

**Programa de Pós-Graduação em**

# Computação Aplicada

**Mestrado Acadêmico**

DARLAN NOETZOLD

Oraculum: A Model for Self-Adaptive System Optimization in Smart Environments

São Leopoldo, 2025

UNIVERSIDADE DO VALE DO RIO DOS SINOS — UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA
NÍVEL MESTRADO

DARLAN NOETZOLD

# ORACULUM: A MODEL FOR SELF-ADAPTIVE SYSTEM OPTIMIZATION IN SMART ENVIRONMENTS

SÃO LEOPOLDO
2025

Darlan Noetzold

# ORACULUM: A MODEL FOR SELF-ADAPTIVE SYSTEM OPTIMIZATION IN SMART ENVIRONMENTS

Dissertation presented as a partial requirement to obtain the Master's Degree from the Applied Computing Graduate Program of the University of Vale do Rio dos Sinos — UNISINOS

Advisor:
Prof. Dr. Jorge Luis Victória Barbosa

Co-advisor:
Prof. Dr. Valderi Reis Quietinho Leithardt

São Leopoldo
2025

N772o Noetzold, Darlan.
Oraculum : a model for self-adaptive system optimization in smart environments / Darlan Noetzold. – 2025.
187 f. : il. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2025.
"Orientador: Prof. Dr. Jorge Luis Victória Barbosa
Coorientador: Prof. Dr. Valderi Reis Quietinho Leithardt."

1. Intelligent environments. 2. Performance metrics. 3. Reinforcement learning. 4. Self-adaptive architecture. 5. Semantic ontology. 6. Sensor data simulation. I. Título.

CDU 004.4

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Silvana Dornelles Studzinski – CRB 10/2524)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior Brasil (CAPES) - Código de Financiamento 001 /This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

*Technology is a useful servant but a dangerous master.*
— CHRISTIAN LOUS LANGE

## ACKNOWLEDGEMENTS

Firstly, I would like to thank my girlfriend Jakelyny for always being by my side, encouraging, supporting me, and giving me the strength to make my dreams come true. Without you, none of this would be possible. I am deeply grateful to my advisors, Prof. Dr. Jorge Luis Victória Barbosa and Prof. Dr. Valderi Reis Quietinho Leithardt, for their dedication, guidance, and knowledge throughout the development of this work. Their trust in my potential and encouragement to advance in the academic world have been invaluable.

I would also like to thank my friends for their partnership and brotherhood, and for understanding my absences during certain periods. My sincere thanks go to the University of Vale do Rio dos Sinos (UNISINOS) for providing the necessary infrastructure and for supporting this research, including the scholarship that made this work possible. Finally, I extend my gratitude to my coworkers and all others who supported me directly or indirectly during this journey.

This work was supported by CNPq (National Council for Scientific and Technological Development—grant number 307137/2022-8), and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil - Finance Code 001).

## ABSTRACT

This dissertation introduces Oraculum, a modular self-adaptive framework designed to support the monitoring, prediction, reasoning, and adaptation of distributed systems operating in smart environments. Many existing solutions treat these tasks as disconnected components, relying on static training phases, fixed adaptation logic, and reactive decision-making triggered only after system degradation is detected. Oraculum proposes an integrated approach in which monitored metrics are continuously collected and processed to generate predictions and select actions in advance of performance failures. The framework consists of three key components. SHiELD is a sensor data simulator that generates synthetic time-series data using ARIMA models and applies heuristic methods-such as filtering, aggregation, and compression-to simulate realistic variability and reduce processing overhead. OntOraculum is a semantic ontology that formalizes performance metrics into five categories and enables the system to classify and validate alerts through rule-based reasoning and SPARQL queries. The adaptation engine uses regression and classification models to forecast short-term metric behavior and integrates a reinforcement learning agent based on a Markov Decision Process (MDP), which receives contextual states and selects actions such as resource scaling, scheduling adjustment, or service reconfiguration. The RL engine also includes a retraining mechanism that periodically updates policies using new data. The entire architecture operates in a closed feedback loop, using predictions and inferred knowledge to support earlier and more informed decisions. The model includes automated pipelines for dataset creation, model training, hyperparameter tuning, and continuous learning, covering both predictive models and RL agents. Experimental validation was conducted in a containerized testbed with simulated load variation. Results were collected across multiple performance indicators, including CPU, memory, latency, and model accuracy. The contributions of this work are: (i) the proposal of an integrated framework that combines monitoring, forecasting, semantic validation, and adaptation; (ii) the development of SHiELD for synthetic data generation and heuristic preprocessing; (iii) the design of OntOraculum for metric classification and rule-based inference; (iv) the implementation of a prediction-based strategy for early alert generation to reduce adaptation delay; and (v) the modeling of an RL engine with configurable actions and scheduled policy retraining.



**Keywords:** self-adaptive architecture, sensor data simulation, semantic ontology, reinforcement learning, performance metrics, intelligent environments..

## RESUMO

Esta dissertação apresenta o Oraculum, um framework modular auto-adaptativo desenvolvido para integrar monitoramento, predição, raciocínio semântico e adaptação em sistemas distribuídos operando em ambientes inteligentes. Muitos trabalhos existentes tratam essas etapas de forma isolada, com processos de treinamento estáticos, lógica de adaptação fixa e decisões reativas que ocorrem apenas após a degradação do desempenho. O Oraculum propõe uma abordagem integrada, na qual métricas monitoradas são processadas continuamente para gerar predições e selecionar ações antes que falhas de desempenho se manifestem. A arquitetura é composta por três componentes principais. O primeiro é o SHiELD, um simulador de dados de sensores que gera séries temporais sintéticas por meio de modelos ARIMA e aplica heurísticas de filtragem, agregação e compressão para simular variabilidade contextual e reduzir o custo de processamento. O segundo é o OntOraculum, uma ontologia semântica que organiza as métricas em cinco categorias e permite a classificação de alertas e a inferência de anomalias por meio de regras SWRL e consultas SPARQL. O terceiro é o mecanismo de adaptação, que utiliza modelos de regressão e classificação para prever o comportamento das métricas e aplica um agente de aprendizado por reforço baseado em um Processo de Decisão de Markov (MDP), capaz de selecionar ações como escalonamento de recursos, ajuste de agendamento ou reconfiguração de serviços. Esse agente também conta com um processo de retreinamento periódico baseado em dados recentes. Toda a arquitetura opera em ciclo fechado, no qual as decisões são antecipadas com base nas predições e inferências. O sistema inclui ainda pipelines automatizados para criação de datasets, treinamento de modelos, ajuste de hiperparâmetros e aprendizado contínuo, tanto para os modelos preditivos quanto para o agente de RL. A validação foi realizada em um ambiente controlado com variações de carga simuladas, e os resultados foram coletados a partir de indicadores como uso de CPU, memória, latência e acurácia dos modelos. As contribuições deste trabalho incluem: (i) a proposta de uma arquitetura integrada que une monitoramento, predição, validação semântica e adaptação; (ii) o desenvolvimento do SHiELD para geração de dados sintéticos e pré-processamento heurístico; (iii) a modelagem do OntOraculum para classificação de métricas e inferência de alertas; (iv) a implementação de uma estratégia de predição para antecipação de alertas e redução de atrasos na adaptação; e (v) o projeto de um motor de aprendizado por reforço com ações parametrizáveis e retreinamento programado.



**Palavras-chave:** arquitetura autoadaptativa, simulação de dados de sensores, ontologia semântica, aprendizado por reforço, métricas de desempenho, ambientes inteligentes..

# List of Figures

# List of Tables

# Contents

# 1 INTRODUCTION

The adoption of smart environments has intensified the need for monitoring solutions capable of addressing the complexity of distributed IoT-based systems. These environments integrate heterogeneous sensors, communication protocols, and processing services, resulting in the continuous production and consumption of large volumes of data. Ensuring the operational stability of these systems depends on the monitoring of critical performance metrics such as memory usage, CPU load, latency, and network throughput (ROSSETTO et al., 2024).

Consider the scenario of a smart city traffic management system. During peak hours, fog computing nodes receive a surge of data from road sensors, surveillance cameras, and air quality monitors. If a sudden increase in vehicle density causes CPU overload and network congestion, traditional monitoring systems-based on fixed thresholds-may delay detection or misclassify the event. Technicians might only react after critical delays impact services, such as failing to reroute traffic in time or losing sensor data due to network failure.

Oraculum addresses this challenge by automating the entire monitoring, learning, and adaptation cycle. Instead of relying on static thresholds or retraining when new metrics are introduced, it continuously processes time-series data, applying regression and classification models to predict critical changes. Adaptation policies are refined through reinforcement learning, enabling the system to act before service degradation occurs. This proactive behavior reduces the average adaptation time, as the system does not wait for explicit failures to occur before acting.

One of the central strengths of the Oraculum architecture is its use of generic performance metrics. Rather than being tailored to a specific metric or application, the model supports a wide range of monitoring data-including hardware, software, network, and contextual information. As a result, new monitored variables can be incorporated into the prediction and adaptation pipelines without requiring reconfiguration or retraining from scratch. This flexibility is critical for smart environments where operational conditions evolve frequently and where scalability is essential.

Traditional monitoring approaches, based on rule-based or threshold-based mechanisms, often fail to anticipate degradations or to support autonomous corrective actions. These methods generally depend on static configurations, where thresholds are predefined without considering the dynamic nature of operational environments. When unexpected variations occur, such as sudden spikes in resource consumption or unpredictable network behavior, static thresholds may either fail to detect the problem or generate false positives, leading to inefficient or delayed responses (COLOMBO et al., 2022). Moreover, they do not incorporate historical behavior patterns or trends, making them inadequate for learning over time. As complexity increases, manually tuning thresholds becomes infeasible, reinforcing the need for predictive and adaptive strategies that operate autonomously and flexibly.

Recent developments have incorporated predictive models to estimate the behavior of performance indicators over time. Regression algorithms process historical time series data to

forecast future metric values, allowing the system to anticipate possible performance issues before service levels are affected (XU; LIU; PAN, 2023). Classification techniques complement regression models by categorizing the state of the system into normal or abnormal behavior, supporting the rapid identification of emerging faults (WEERASINGHE et al., 2024).

To strengthen autonomous system behavior, reinforcement learning has been increasingly applied in monitoring architectures. Unlike static or predefined adaptation strategies, reinforcement learning enables systems to improve their decision-making processes based on continuous environmental interaction. Learning agents receive feedback from monitoring components and adjust their actions by maximizing long-term rewards, rather than reacting solely to immediate conditions. This characteristic allows the system to balance short-term performance gains with long-term operational stability. Agents can explore different adaptation strategies, learn from previous outcomes, and gradually refine their policies to better handle complex and dynamic scenarios. They are able to interact with multiple layers of the system architecture, responding to different alert types generated by predictive or classification models, and dynamically adjusting operational parameters such as resource allocation, scaling strategies, or configuration settings (MADHUNALA; ANANTHA, 2022). Through this continuous learning cycle, reinforcement learning supports the development of systems capable of adapting to unforeseen changes and maintaining desired performance levels even in volatile and heterogeneous environments.

For instance, in a smart hospital environment, Oraculum can be used to monitor and manage the IoT infrastructure supporting critical medical devices such as heart monitors, infusion pumps, and patient tracking systems. During emergency situations, such as a sudden increase in patient admissions in a specific ward, the system can predict network overloads and identify potential processing bottlenecks in real time. By anticipating these events, Oraculum can trigger adaptation policies like automatically redistributing workloads across servers, prioritizing sensitive data traffic, or dynamically adjusting computational resources. This ensures the continuity and quality of medical services, minimizing the risk of failures and optimizing operational response in highly dynamic and critical environments.

Similarly, in a hybrid e-commerce scenario that integrates both online and physical store operations, Oraculum can monitor the performance of interconnected systems such as inventory management, point-of-sale terminals, and customer analytics platforms. For example, during a major sales event, a surge in both online orders and in-store purchases may strain backend systems and network resources. Oraculum can proactively detect early signs of resource saturation or abnormal transaction latency, enabling the system to automatically scale cloud resources, balance network loads, or adjust in-store device configurations. This proactive adaptation helps maintain seamless customer experiences across digital and physical channels, prevents service disruptions, and supports efficient, data-driven retail operations.

Although research has advanced in predictive monitoring and reinforcement learning, comprehensive frameworks that integrate regression, classification, and adaptive decision-making

into a unified structure remain limited. Many existing solutions address these capabilities in isolation, without fully connecting predictive analysis to autonomous adaptation strategies (STEHLE et al., 2024). There remains a significant opportunity to develop scalable, modular models that manage real-time metrics, support accurate forecasting, and enable autonomous system adjustments in response to predicted conditions.

## 1.1 Motivation

The development of smart environments involves multiple challenges, including the simulation of sensor data, the processing of real-time information, and the implementation of adaptive mechanisms capable of responding to dynamic conditions. Existing IoT simulators tend to focus on isolated aspects, such as network topology, communication performance, or scalability evaluation. However, they often do not simulate dynamic sensor behavior or provide integrated support for prediction, anomaly detection, and self-adaptation in response to evolving operational contexts (HU et al., 2023).

To address these limitations, the Oraculum model proposes a unified architecture that integrates sensor data simulation, predictive modeling, classification of system states, and reinforcement learning-based adaptation. By combining these components, the model supports the continuous monitoring of performance metrics, anticipates anomalous behaviors, and applies corrective actions in real time. This integration allows for a more realistic evaluation of smart environments, where data variability, system constraints, and adaptation needs coexist and influence overall performance. Oraculum aims to provide a complete environment for testing and validating self-adaptive strategies across multiple layers of the system, offering insights into the behavior of intelligent distributed infrastructures under dynamic and uncertain conditions.

Recent research has investigated the application of reinforcement learning and predictive techniques in smart environments, particularly in energy management, smart cities, and sensor networks. For instance, SOURI et al. (2022) presents a comprehensive review of reinforcement learning approaches in smart environments, highlighting the main learning models and their adaptation to different applications. The authors emphasize how Q-learning and Deep Q-Networks (DQNs) have been adopted to manage resources dynamically in smart grids and IoT scenarios, with impacts on energy consumption and task scheduling efficiency.

In addition, (ZANELLA GOMES et al., 2019) provides a detailed survey on the use of reinforcement learning in the Internet of Things (IoT), discussing key challenges and the role of RL agents in autonomous decision-making. The study categorizes applications into areas such as network optimization, energy efficiency, and latency reduction, and points out the need for hybrid models that combine supervised learning and RL to improve adaptability. These findings support the motivation for the proposed architecture, which integrates predictive models with reinforcement learning to enable real-time monitoring and adaptive actions in distributed sensor-based environments.

Furthermore, adaptive systems that incorporate reinforcement learning or other intelligent methods require environments that respond to agent actions and allow changes in state representations. Traditional simulators do not offer these adaptive interaction cycles, which are necessary for experiments involving self-adaptive software architectures (FORTINO; SAVAGLIO; ZHOU, 2019).

Interoperability also represents an important constraint. Simulators frequently support a limited set of communication protocols, reducing their applicability to scenarios involving heterogeneous devices and decentralized communication through technologies such as MQTT and ZigBee (ALMUTAIRI; BERGAMI; MORGAN, 2024a). These gaps restrict the potential for validating solutions designed for real-world deployment.

In addition to these limitations, many of the architectures presented in the literature are designed for specific application domains. For example, the model in (SAH et al., 2022a) focuses on energy and traffic metrics but does not apply learning techniques or offer predictive capabilities, relying solely on reactive responses. Architectures such as (ETEMADI; GHOBAEI-ARANI; SHAHIDINEJAD, 2021) and (YANG et al., 2021a) apply supervised learning for regression or anomaly classification but treat training as a one-time process, without integrating mechanisms for automatic dataset updates or model retraining. This limitation affects model accuracy over time, especially in environments with dynamic workloads.

Works that apply reinforcement learning, like (TAM; MATH; KIM, 2022a) and (CEN; LI, 2022a), rely on reactive adaptation. The agent begins decision-making only after performance degradation is detected, which increases the response time and limits the effectiveness of the adaptation. In many of these cases, the monitored metrics are restricted to predefined sets-such as throughput, delay, or CPU usage-and there is no abstraction layer to generalize the adaptation logic across different contexts.

Additionally, some architectures with broader metric monitoring coverage, like (SAMARAKOON et al., 2023a), do not integrate AI models, which limits their ability to act autonomously in dynamic scenarios. Although models like (VELRAJAN; SHARMILA, 2023) employ reinforcement learning across the full MAPE cycle, they do not implement prediction-based alerts, which would allow earlier decision-making.

In response to these challenges, this work presents a model architecture designed to address the integration of synthetic sensor data generation, real-time processing, and adaptive decision-making based on predictive models and reinforcement learning. The model enables the emulation of smart environments with temporal variation, supports modules for anomaly detection and metric forecasting, and provides interfaces for simulating actions that intelligent agents may take under different operational conditions. In addition, the architecture includes automated pipelines for data collection, model training, and policy retraining, improving adaptability and enabling continuous learning without manual intervention.

## 1.2 Problems and Questions

The growing complexity of smart environments has increased the demand for solutions capable of supporting autonomous decision-making based on real-time sensor data. The heterogeneity of devices, the variability in sensor data quality, and the unpredictability of contextual changes present significant challenges to maintaining performance and reliability. Current approaches for monitoring and adaptation often fail to scale efficiently, lack predictive capabilities, or rely heavily on manual interventions, reducing the system's overall resilience and adaptability. Moreover, the combination of historical context and predictive analytics remains underexplored in scenarios where proactive and automated responses are crucial.

In this sense, a strategy to improve performance and reduce failures in smart environments involves integrating predictive models and reinforcement learning into a unified computational architecture. The proposed model, Oraculum, is designed to capture sensor data, predict future states, and apply adaptive actions based on contextual patterns and performance metrics. Oraculum uses time-series monitoring, ontology-based inference, and reinforcement learning agents to determine optimal responses in complex environments. Historical data is leveraged to detect oscillations in behavior, anomalies in performance, and potential intermittent failures, allowing the system to anticipate problems and react appropriately. Therefore, this study is guided by the following general research question: "How can the integration of monitoring, prediction, and reinforcement learning support adaptive behavior in smart environments, ensuring system resilience and performance?". Specific questions have also been defined to support the answer to the general question, as follows:

- What are the main performance metrics and contextual parameters required to ensure adaptation in smart environments?
- How can predictive models based on time-series data be used to anticipate failures or performance degradation?
- How can ontological reasoning contribute to the contextual understanding of sensor data in dynamic environments?
- What is the role of reinforcement learning in supporting autonomous decisions within a distributed sensor architecture?
- How can historical context patterns improve the identification of anomalies and trigger adaptive actions?
- What types of actions can be defined and executed to optimize system performance in response to predicted issues?
- How does the integration of monitoring, prediction, and adaptation in a unified architecture impact the system's resilience and efficiency?

## 1.3 Objectives

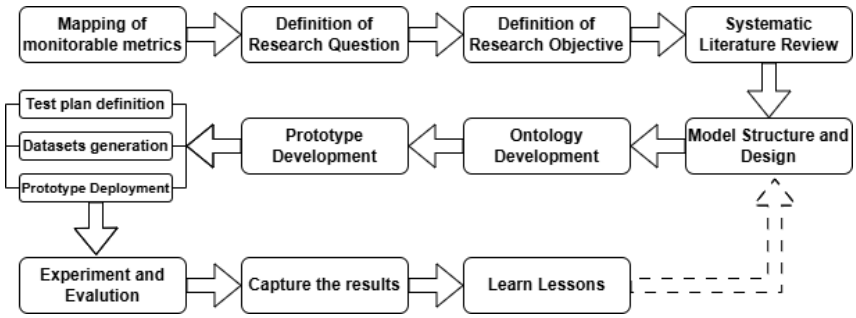
The main objective of this study is to develop a computational model, named Oraculum, that reduces the average adaptation time in smart environments while preserving adaptation efficiency. To achieve this, the model integrates predictive strategies based on time-series analysis and reinforcement learning. Oraculum aims to anticipate performance deviations and apply corrective actions before service degradation occurs, thus improving responsiveness and system resilience. By combining metric prediction, ontological reasoning, and adaptive control, the model seeks to operate in a fully automated manner, supporting continuous and context-aware adaptation. The following specific objectives support the achievement of this general goal:

- Reduce the Mean Adaptation Time (MAT) to values close to zero, enabling prompt responses to changes in the monitored environment.
- Validate the proposed model through a functional prototype capable of monitoring performance metrics from at least three distinct domains, including software, hardware, and network, with potential extension to SLA-related indicators.
- Maintain Adaptation Accuracy (AA) above 90%, ensuring the correct selection of actions under varying operational conditions.
- Keep Adaptation Overhead (AO) below 10%, minimizing the impact of adaptation mechanisms on system performance.
- Ensure Adaptation Stability (AS) above 85%, reflecting the model's ability to preserve consistent behavior over time in the presence of dynamic events.
- Develop at least one regression model capable of forecasting metric values up to two minutes ahead, achieving a Normalized Root Mean Square Error (NRMSE) above 85%.
- Train classification models that reach an accuracy of at least 90%, allowing effective detection of anomalous or degraded states.
- Automate the entire pipeline, including data collection, dataset construction, training and revalidation of regression and classification models, hyperparameter tuning, alert triggering, and execution of adaptation actions. The only manual tasks involve the definition of adaptation parameters and the reward tuning for the reinforcement learning agent.
- Release the OntOraculum metric ontology as a reusable and documented resource for other intelligent monitoring projects, ensuring proper structure and alignment with the proposed architecture.
- Achieve stable convergence of the reinforcement learning agent, verified by consistent decision-making patterns over time in various environmental scenarios.

## 1.4 Methodology

Figure 1 illustrates the research methodology adopted in this dissertation. The first stage of this study consisted of identifying and analyzing domains related to smart environments, performance metrics, time-series data, sensor networks, and adaptive systems. This stage enabled the mapping of monitorable metrics, followed by the definition of the research question and the formalization of the research objective, which guided the entire development of the Oraculum model.

Figure 1: Research flow adopted in the development of the Oraculum model.




**Source:** Elaborated by the author

Subsequently, a systematic literature review was conducted to identify the current state of the art, existing gaps, and promising approaches in the areas of performance monitoring, machine learning, ontologies, and reinforcement learning applied to smart environments. This review supported the theoretical foundation of the research and reinforced the motivation for developing a novel integrated model.

The structure and design of the Oraculum model were proposed based on the information collected and the knowledge gaps identified. This step detailed the architectural components, data flow, expected functionalities, and required technologies. In parallel, the development of an ontology was carried out to organize the data representation and enable logical inferences about the performance and contextual information obtained from sensors.

In the following step, a prototype of the Oraculum model was implemented. This stage included defining the test plan, generating synthetic datasets for evaluation, and deploying the prototype. The datasets were produced by a dedicated simulation component, ensuring diversity and consistency across different use cases and performance scenarios.

The prototype was then used in the experimentation and evaluation phase, in which the Oraculum model was tested in real or simulated environments. During this phase, the integration of data fusion, predictive models, and reinforcement learning agents was validated. The results of the experiments were analyzed and documented to assess the effectiveness of the model in optimizing system behavior, detecting anomalies, and applying appropriate adaptations.

Finally, the research process culminated in the stage of capturing results and learning lessons, where the observations were used to refine the Oraculum model. This cycle allowed continuous improvement of the ontology, the ML models, and the reinforcement learning strategies used by the system.

## 1.5 Contributions

The main scientific contribution of this research is the development of the Oraculum model, a modular and extensible framework that integrates metric monitoring, predictive modeling, semantic reasoning, and reinforcement learning to support proactive and autonomous adaptation in smart environments. The model combines knowledge representation with data-driven learning techniques to anticipate system degradation and execute timely corrective actions. A key objective addressed by the framework is the reduction of adaptation time while maintaining adaptation quality, minimizing resource overhead, and ensuring behavioral stability during adaptation cycles. The specific contributions of this work are outlined below:

1. OntOraculum ontology and semantic alert validation: This contribution includes the design and implementation of the OntOraculum ontology, which formally structures metrics, contextual states, and system entities. The ontology enables semantic classification of monitored data and supports alert inference through SWRL rules and SPARQL queries. It is used to assist the creation of labeled datasets and to validate at runtime whether a predicted anomaly corresponds to a relevant adaptation trigger.

2. Delay reduction strategy through metric forecasting and proactive alerting: The model introduces a prediction-based strategy to reduce adaptation delays. By forecasting future values of monitored metrics and anticipating critical thresholds, the system proactively triggers alerts before performance degradation occurs. This approach allows the reinforcement learning agent to make informed decisions with additional lead time, contributing to more timely and stable adaptations.

3. MDP-based reinforcement learning engine with scheduled retraining and parameterized actions: The reinforcement learning component is formulated as a Markov Decision Process (MDP), incorporating parameterized actions such as vertical and horizontal scaling, service restarts, and heuristic adjustments. The RL engine supports scheduled retraining using updated context data and maintains adaptation strategies aligned with evolving system behavior. It is responsible for decision-making under uncertainty, aiming to reduce adaptation time while sustaining adaptation accuracy and low overhead.

4. Automated Architecture for Data Set Generation, Model Training, and Hyperparameter Tuning: This contribution includes a fully automated pipeline for collecting sensor data,

constructing datasets, training predictive models, and tuning hyperparameters. Both regression and classification models are generated and periodically updated per metric and per system node. The same pipeline is used to manage the retraining of the reinforcement learning agent, ensuring consistent alignment with system dynamics.

## 1.6 Outline

This dissertation is organized into nine chapters. Chapter 2 presents the theoretical foundations necessary for the development of the Oraculum model, covering topics such as system adaptation, metric monitoring, MAPE-K architectures, ontologies, and artificial intelligence techniques. Chapter 3 provides a systematic literature review, analyzing how self-adaptive systems are designed in smart environments and identifying research gaps and opportunities.

Chapter 4 introduces the SHiELD simulator, developed to generate synthetic data and validate adaptation strategies. Chapter 5 presents the OntOraculum ontology, detailing its development, structure, and use in anomaly inference. Chapter 6 describes the Oraculum model proposal, including its architecture and core components. Chapter 7 focuses on the implementation aspects, such as the application of predictive models, the reinforcement learning agent, and the system modules. Chapter 8 discusses the experiments conducted, the results obtained across system layers, and the performance evaluation of the architecture. Finally, Chapter 9 presents the final considerations, highlighting the contributions, limitations, and future research directions of this work.

# 2 BACKGROUND

This chapter presents the core concepts that provide the theoretical foundation for the development of the Oraculum model. The presented concepts encompass multiple areas that are essential to understanding and constructing adaptive monitoring and prediction systems for smart environments. The first section explores system adaptation mechanisms and their importance in dynamic and heterogeneous environments. The second section discusses the monitoring of metrics, highlighting the types of data collected and their role in decision-making processes. The third section introduces MAPE-K systems as a conceptual model for autonomous computing, guiding system adaptation through monitoring, analysis, planning, execution, and knowledge components. The fourth section covers ontologies, focusing on their role in organizing and reasoning over contextual information. The final section addresses Artificial Intelligence, with emphasis on relevant techniques such as machine learning and reinforcement learning, which enable predictive analysis and decision-making within the Oraculum architecture.

## 2.1 Systems Adaptation

System adaptation enables dynamic adjustment of computational behavior in response to environmental or workload changes. This capability is essential in smart environments composed of heterogeneous and distributed systems. Reinforcement learning has been applied to minimize energy consumption and improve Age of Information (AoI) in wireless energy transfer systems (XU; LIU; PAN, 2023), as well as to optimize context caching in real-time distributed applications (WEERASINGHE et al., 2024). Adaptive orchestration strategies in fog computing environments support SLA preservation and trigger self-healing mechanisms (COLOMBO et al., 2022).

DeepHYDRA integrates deep neural networks with clustering for real-time anomaly detection in high-performance systems (STEHLE et al., 2024). In smart buildings, federated learning supports decentralized anomaly detection while maintaining data privacy (SATER; HAMZA, 2021).

Task scheduling and adaptive resource management approaches have been employed in fog-cloud environments to optimize QoS and cost (HOSEINY et al., 2021). In industrial settings, distributed QoS-aware routing schemes reduce latency and packet loss (S; KANNIGA, 2023), and integrated operational monitoring improves infrastructure efficiency (XUE et al., 2023).

Vehicular fog computing leverages real-time offloading strategies to meet delay constraints (WU et al., 2023), and scalable aggregation techniques enable adaptation in large sensor networks (HENNING; HASSELBRING, 2019). Efficient monitoring strategies for fog architectures support resource adaptation in distributed contexts (BATTULA et al., 2020).

## 2.2 Monitoring Metrics

Monitoring in smart environments enables adaptive systems to detect deviations, anticipate failures, and support resource-aware control. In IoT infrastructures, performance metrics are used not only for observation but for triggering real-time decisions.

Low-power monitoring techniques are critical in constrained environments. Adaptive sensor nodes designed with deep learning and in-field programmability allow runtime reconfiguration (SHAFIEE; OZEV, 2022). Efficient access methods without prior reservation improve sustainability and connectivity in dense IoT networks (AZARI et al., 2021). These strategies reduce energy waste while maintaining data fidelity.

In industrial settings, resource usage monitoring enables real-time anomaly detection in NoSQL databases (CHOULIARAS; SOTIRIADIS, 2020), while edge-based learning models have been applied for fault detection in power distribution (WEI et al., 2022). Highway infrastructures integrate monitoring with predictive models to improve road safety through sensor data and intelligent feedback (SINGH et al., 2021).

Residential energy systems use metric-based optimization for managing energy tasks and scheduling (IMRAN; IQBAL; KIM, 2022). In smart grids, AI frameworks improve the efficiency of data acquisition and consumption, supporting predictive load balancing (LI et al., 2024a). These examples confirm that monitored metrics guide proactive responses across energy-centric domains.

Security applications rely on monitoring for behavior classification and threat identification. Deep learning models enriched by MUD policies support anomaly detection in IoT devices (MIRDULA; ROOPA, 2023), while smart construction systems combine vision and metric monitoring to enhance site automation (BADUGE et al., 2022).

Time series data from sensors is used in forecasting models to anticipate demand. ConvLSTM networks have been used to predict household electricity consumption, aligning supply with expected usage (CASCONE et al., 2023). In solar-powered wireless sensor networks, adaptive broadcasting policies depend on continuous energy harvesting metrics to balance communication and power use (KHIATI; DJENOURI, 2018).

These contributions show that monitoring has evolved from a passive observation tool into an active component of adaptive systems, enabling prediction, anomaly detection, and control in real time.

## 2.3 MAPE-K Systems

The MAPE-K model structures adaptive behavior through monitoring, analysis, planning, execution, and a shared knowledge base. It is used to coordinate adaptation in distributed systems and IoT environments.

In energy management, the MAPE-K loop supports forecasting and load optimization. IoT-

enabled platforms integrate monitoring and planning to improve PV power generation and load balancing (RAO; SAHOO; YANINE, 2024). Reinforcement learning complements this process by provisioning microservice-based IoT devices in dynamic contexts (RATH; MANDAL; SARKAR, 2024). Adaptive optimization techniques further refine energy efficiency, as seen in the application of metaheuristics and improved neural networks (NANDISH; PUSHPARAJESH, 2024; REVANESH et al., 2023).

Monitoring and planning are integrated into service orchestration. Real-time frameworks using services like AWS IoT and DeepAR enable predictive management in industrial environments (CHAKRABARTI; SADHU; PAL, 2023). Scheduling approaches combine machine learning and rule-based planning for dynamic resource allocation in cloud platforms (ZHOU, 2023). In telecommunications, energy feedback systems rely on continuous monitoring for decision support (SORRENTINO; FRANZESE; TRIFIRÒ, 2024).

Fog and edge computing architectures use the MAPE-K loop to manage latency and resource distribution. Advanced orchestration strategies optimize task placement in heterogeneous layers (SRICHANDAN et al., 2024). Anomaly detection is enhanced through explainable LSTM-based models, improving observability and interpretability in fog networks (SHARMA; KAUR, 2023).

The knowledge and monitoring components of MAPE-K are often supported by scalable infrastructures. Platforms using Elasticsearch and Kafka provide observability in IoT deployments (CALDERON et al., 2023). Load balancing in fog environments is reinforced by reinforcement learning policies trained through performance feedback (TALAAT et al., 2020). Adaptive caching and data compression mechanisms also rely on performance monitoring to adjust strategies dynamically (MEENA et al., 2023; SURESHKUMAR; SABENA, 2023).

Multi-agent systems extend the MAPE-K framework through distributed learning. MARL protocols optimize routing and task execution in sensor networks PRABHU; ALAGESWARAN; MIRUNA JOE AMALI (2023), while genetic algorithms and neural approaches improve energy-aware allocation strategies (AZEVEDO ALBUQUERQUE et al., 2023; PRASANNA et al., 2023). These applications illustrate the continued relevance of MAPE-K as a coordination structure for intelligent, adaptive systems.

## 2.4 Ontologies

Ontologies formalize domain knowledge and support semantic interoperability in distributed and intelligent systems. They define concepts, relationships, and constraints, enabling reasoning and contextual inference (GRUBER, 1995; GUARINO, 1998; NOY; MCGUINNESS et al., 2001).

In monitoring architectures, ontologies are used to structure metrics and support performance reasoning. DAEMON applies semantic descriptions to IoT monitoring environments DAOUDAGH; MARCHETTI (2023), and other works model performance indicators for build-

ings and grid workflows (CORRY et al., 2015; TRUONG et al., 2005). Ontologies also support testing automation and guide performance evaluation in distributed applications (FREITAS; VIEIRA, 2014).

Service-oriented architectures benefit from semantic enrichment. Ontologies enhance service discovery and selection by modeling quality attributes and user preferences (BAOCAI et al., 2010; CAO et al., 2019). In cloud and edge computing, ontologies organize functional and non-functional service attributes to support classification and resource selection (AL-SAYED; HASSAN; OMARA, 2020; METWALLY; JARRAY; KARMOUCH, 2015).

Resource management strategies often rely on the evaluation of semantic rules. Ontology-based frameworks are applied in decision-making platforms and user-defined policy evaluation (YASEEN et al., 2011). Multi-agent systems also use ontologies to coordinate distributed resource planning and adaptive behavior (RZEVSKI; SKOBELEV; ZHILYAEV, 2022).

The development of ontologies is supported by tools such as Protégé MUSEN (2015a,b) and rule languages like SWRL HORROCKS et al. (2004); O'CONNOR; DAS (2005). Recent tools integrate deep learning with ontological reasoning, such as DeepOnto BLOMQVIST et al. (2024).

Sensor ontologies address interoperability in heterogeneous environments, aligning models through swarm-based alignment techniques (LATEEF HAROON P S et al., 2024). Domain-specific ontologies are applied in healthcare monitoring SHARMA et al. (2021); ZESHAN et al. (2023a), service modeling MARIć; BACH; GUPTA (2024), and energy management IMRAN; IQBAL; KIM (2022); CHAKRABARTI; SADHU; PAL (2023); RAO; SAHOO; YANINE (2024). These applications demonstrate the role of ontologies in structuring performance knowledge, supporting reasoning over system behavior, and integrating metrics into self-adaptive decision-making models.

## 2.5 Artificial Intelligence

Artificial Intelligence (AI) has become a key enabler in the design and operation of smart environments, providing the foundation for predictive analytics, adaptive behavior, and autonomous decision-making. Its integration into Internet of Things (IoT) architectures allows for enhanced data processing, real-time monitoring, and dynamic system optimization. AI techniques support a variety of computational tasks, including forecasting, classification, anomaly detection, and resource management, making them suitable for complex and heterogeneous infrastructures.

In the context of smart systems, different AI paradigms are employed to address specific challenges. Regression models are commonly used for predicting future values of continuous variables such as energy consumption or system load. Classification models enable the detection of patterns, faults, and critical events through labeled datasets. Reinforcement Learning (RL), on the other hand, allows agents to interact with the environment and improve performance over

time through experience-based learning. Each of these AI branches brings distinct capabilities and limitations, which are explored in the subsections that follow.

### 2.5.1 Regression Models

Regression models are essential in intelligent systems for predicting continuous variables based on historical and contextual data. These models are used to forecast energy consumption, temperature, latency, traffic load, and many other quantitative metrics. In contrast to classification, which assigns inputs to discrete classes, regression aims to model a mapping $f : \mathbb{R}^n \to \mathbb{R}$ that minimizes the prediction error over a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$.

The most fundamental approach is linear regression, which assumes the output variable is a linear combination of input features:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$$

The optimal parameters $\mathbf{w}$ and $b$ are estimated by minimizing a loss function, commonly the Mean Squared Error (MSE):

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

This approach was adopted by MALEKZADEH (2023) to model and predict the performance of 5G communication networks, improving system responsiveness through early identification of performance degradation trends.

In energy-related applications, RAO; SAHOO; YANINE (2024) developed a forecasting model for photovoltaic (PV) power generation using regression, which allowed for optimized load management in IoT-enabled smart grids. Accurate predictions in such contexts are crucial to avoid overproduction or underutilization of energy resources.

To capture nonlinear relationships and temporal dependencies, deep learning-based regression has gained prominence. CASCONE et al. (2023) proposed a hybrid approach using Convolutional LSTM (ConvLSTM) networks for multi-step household energy forecasting. The model integrates convolutional layers for spatial pattern extraction with LSTM units for temporal modeling, enabling effective prediction in complex, high-dimensional time series environments.

Time series regression models often aim to predict a sequence of future values $\{y_{t+1}, y_{t+2}, ..., y_{t+h}\}$ given past observations $\{y_t, y_{t-1}, ..., y_{t-p+1}\}$. The prediction can be formulated as:

$$\hat{y}_{t+k} = f(y_t, y_{t-1}, ..., y_{t-p+1}) \quad \text{for } k = 1, ..., h$$

where $h$ is the forecasting horizon and $p$ is the window size of the past context.

More complex techniques, such as Support Vector Regression (SVR), decision tree regres-

sors (e.g., XGBoost, LightGBM), and ensemble models, have also been widely applied, especially when dealing with nonlinear patterns and high-dimensional feature spaces.

Evaluation metrics for regression include:

- Mean Absolute Error (MAE): $\frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$
- Root Mean Squared Error (RMSE): $\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$
- Coefficient of Determination ($R^2$):

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

These metrics guide the selection and tuning of models, ensuring accurate and reliable forecasting in smart environments. As intelligent systems scale in complexity and heterogeneity, regression models remain a cornerstone for proactive and adaptive management of continuous-valued metrics.

### 2.5.2 Classification Models

Classification models play a fundamental role in intelligent environments, particularly for tasks such as anomaly detection, occupancy recognition, fault classification, and security monitoring. These models learn to map an input feature vector $\mathbf{x} \in \mathbb{R}^n$ to one of several predefined classes $y \in \{1, 2, ..., K\}$, where $K$ denotes the number of categories. Given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, the goal is to learn a function $f : \mathbb{R}^n \to \{1, ..., K\}$ that generalizes well to unseen data.

One of the most widely used paradigms in classification is the use of probabilistic discriminative models, which estimate the posterior distribution $P(y \mid \mathbf{x})$. For example, in softmax classifiers, the output probabilities are given by:

$$P(y = k \mid \mathbf{x}) = \frac{e^{\mathbf{w}_k^\top \mathbf{x}}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^\top \mathbf{x}}}$$

where $\mathbf{w}_k$ is the weight vector corresponding to class $k$. This formulation is also the basis for the final layer in many neural network architectures used for classification.

Neural networks, particularly recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), are increasingly applied in classification tasks with critical temporal dependencies. For instance, SHARMA; KAUR (2023) proposed XLAAM, an explainable LSTM-based model designed to classify user behaviors and detect anomalies in fog-based environments. The architecture integrates LSTM units with attention mechanisms and interpretable components to enhance explainability and performance in dynamic systems.

In wireless sensor networks (WSNs), classification is often constrained by energy and computational budgets. REVANESH et al. (2023) addressed this by proposing an improved Levenberg–Marquardt neural network architecture, enhancing classification accuracy while preserving energy efficiency. This is particularly important for real-time monitoring applications in smart cities and smart grids.

Classification in IoT security is another critical application. MIRDULA; ROOPA (2023) presented a deep learning-based classification framework that incorporates manufacturer usage descriptions (MUD) to classify network behaviors of IoT devices, enabling threat detection and mitigation in smart buildings.

Classical models like decision trees, support vector machines (SVMs), and Naïve Bayes classifiers are still used, especially when interpretability and low-latency inference are required. For multi-class problems, these models can be extended through strategies like one-vs-rest (OvR), one-vs-one (OvO), or through probabilistic generative formulations such as Gaussian Mixture Models (GMMs).

To further evaluate classification models, metrics such as precision, recall, F1-score, and the confusion matrix are commonly employed. For a binary classification task, the F1-score is computed as:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2TP}{2TP + FP + FN}$$

where $TP$, $FP$, and $FN$ denote true positives, false positives, and false negatives, respectively.

Recent trends involve the use of hybrid models and ensemble learning techniques that combine multiple classifiers to improve robustness and accuracy. Moreover, transfer learning and pre-trained models (e.g., BERT, ResNet) have been increasingly adopted in sensor-rich environments, particularly when annotated data is scarce. The diversity of classification techniques enables flexible adaptation to a wide range of contexts in intelligent systems, from real-time anomaly detection to high-dimensional behavioral analysis.

### 2.5.3 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where agents learn optimal behaviors by interacting with an environment to maximize cumulative rewards, typically modeled as a Markov Decision Process (MDP) defined by the tuple $(S, A, P, R, \gamma)$, where $S$ is the set of states, $A$ the set of actions, $P$ the transition probability, $R$ the reward function, and $\gamma$ the discount factor.

The agent seeks a policy $\pi : S \to A$ that maximizes the expected return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Key value functions are:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right], \quad Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s, A_t = a\right]$$

Their recursive Bellman equations are:

$$V^{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

$$Q^{\pi}(s,a) = \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') Q^{\pi}(s',a')]$$

Common solution methods include policy iteration, value iteration, and Q-learning. Advanced approaches, such as policy gradients and actor-critic methods, optimize parameterized policies in continuous spaces, with the policy gradient given by:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}}\left[\nabla_{\theta} \log \pi_{\theta}(a \mid s) Q^{\pi_{\theta}}(s,a)\right]$$

The integration of deep learning and RL, known as deep reinforcement learning (DRL), enables the use of neural networks to approximate value functions or policies, making RL applicable to complex, high-dimensional environments.

Reinforcement learning has demonstrated significant utility in intelligent environments. TALAAT et al. (2020) proposed a load balancing and optimization strategy using RL to manage resource allocation in fog computing dynamically. PRABHU; ALAGESWARAN; MIRUNA JOE AMALI (2023) developed a multi-agent reinforcement learning (MARL) framework to improve energy-efficient routing in wireless sensor networks. In the Internet of Medical Things (IoMT), NAZARI et al. (2023) introduced a reinforcement learning-based routing protocol focused on Quality of Service (QoS) and energy awareness.

Additionally, RATH; MANDAL; SARKAR (2024) employed a context-aware RL algorithm to provision devices dynamically in microservice-based IoT architectures. ESFAHANI; ELKHODARY; MALEK (2013) presented a learning-based framework tailored for engineering self-adaptive software systems, integrating RL techniques to support runtime adaptation through feature-oriented models.

# 3 LITERATURE REVIEW

This chapter presents a structured review of the literature on techniques and tools for monitoring performance and enabling self-adaptation in smart environments. As systems become increasingly complex and operate under dynamic and uncertain conditions-particularly in domains such as cloud computing, the Internet of Things (IoT), and smart cities-the ability to autonomously adapt to contextual changes has become a fundamental design requirement.

Modern software systems must ensure continuous operation while responding to fluctuating workloads and environmental variations. For example, cloud-based services need to maintain service quality and cost-effectiveness despite abrupt shifts in demand. These challenges have motivated the development of monitoring mechanisms and adaptive strategies that enable systems to reconfigure themselves at runtime to meet operational objectives (SOURI et al., 2022; ALKHAYYAL; MOSTAFA, 2024).

While conventional solutions such as exception handling and fault tolerance remain in use, they are often tightly integrated with application logic, which limits their generalization and reuse (GARLAN et al., 2004). In contrast, external adaptation mechanisms based on feedback loops support modularity and flexibility, offering a more scalable and decoupled approach to self-adaptation. Within this context, self-adaptive systems are defined by their capacity to adjust behavior autonomously based on monitoring data and decision-making models (YANG et al., 2021b). These models include reinforcement learning (TALAAT et al., 2020; PRABHU; ALAGESWARAN; MIRUNA JOE AMALI, 2023), predictive analytics, and search-based techniques such as evolutionary programming (IMRAN; IQBAL; KIM, 2022; ZHOU, 2023), which support the decision-making process in real time.

Recent literature also emphasizes the role of adaptive strategies in urban governance and critical infrastructures, highlighting the broader relevance of these approaches in both technical and societal contexts (ABU-TAYEH et al., 2023; CARDULLO; KITCHIN, 2024; SUSHA; GIL-GARCIA, 2023; JANSSEN; KUK, 2024). In line with this perspective, the present research conducts a systematic review aimed at identifying and analyzing existing approaches that contribute to the development of the proposed model. A total of 133 peer-reviewed studies were selected from seven academic databases, enabling the identification of central research themes, technological directions, and unresolved challenges in monitoring and self-adaptation. This analysis organizes and categorizes the techniques and tools used for monitoring and adaptive responses in smart environments, outlines the different types of smart environments addressed in the literature, and classifies the main performance metrics and frequent bottlenecks encountered in such systems. Additionally, the review maps the adaptation and monitoring strategies applied to runtime systems and synthesizes relevant studies that provide both methodological guidance and conceptual support for this work. Grounded in these insights, the research builds a solid theoretical and methodological basis for the design of the proposed solution.

## 3.1 Research Methodology

This study adopts the systematic literature review approach proposed by PETERSEN et al. (2008) to investigate how monitoring and self-adaptation techniques are applied in smart environments. Systematic reviews provide a structured and objective method for synthesizing existing research, identifying gaps, and recognizing trends across studies. Following the methodological guidelines defined by PETERSEN et al. (2008), this process defines research questions, search procedures, selection criteria, and conducts analysis and classification of the selected studies. This framework allows for a consistent evaluation of how various techniques are implemented, which performance metrics are monitored, and what common system bottlenecks are reported in the literature.

### 3.1.1 Research questions

Table 1 presents the research questions, consisting of a general question (GQ), five specific questions (FQ), and one statistical question (SQ). The general question addresses how studies apply monitoring and self-adaptation techniques in smart environments, both with and without the use of machine learning. The specific questions focus on particular challenges, technologies, techniques, performance metrics, and additional resources used in these studies. Finally, the statistical questions assess the publication sources of the articles and the number of publications per year.

Table 1: Research questions

| Reference | Question |
|---|---|
| **General Question** | |
| GQ1 | How are performance metrics being monitored and systems being self-adapted in studies related to Smart Environments? |
| **Focused Questions** | |
| FQ1 | What techniques and tools are being used to monitor specific performance metrics in smart environments? |
| FQ2 | What types of smart environments are covered in the reviewed studies? |
| FQ3 | What specific performance metrics are being monitored in the studies? |
| FQ4 | What are the main bottlenecks or challenges identified in the applications of these environments? |
| FQ5 | How are Machine Learning techniques being used to enable self-adaptation in the reviewed systems? |
| **Statistical Questions** | |
| SQ1 | Where were the studies published and how has the number of publications evolved per year? |

**Source:** Elaborated by the author

### 3.1.2 Search process

Table 13 presents the search string combining terms related to performance metrics, smart environments, and machine learning techniques, which are the main themes of this study. The first part of the string contains terms related to performance metrics, such as "performance metrics" and "monitoring management." The second part focuses on smart environments, including

terms like "smart environment" and "IoT." The third part covers adaptation techniques and monitoring technologies, such as "machine learning," "reinforcement learning," and "self-adaptive."

The search process requires that selected studies contain at least one term from each part of the search string. This rule was applied to restrict the scope of the research, as initial searches without specific terms returned an overly broad set of articles. Furthermore, the initial searches helped define specific terms that cover a range of scenarios within smart environment research. At this stage, articles that did not directly focus on smart environments and self-adaptation were gathered. The focus on self-adaptation techniques and continuous monitoring was established in the final phases of this research.

After defining the search string, the databases to be analyzed were selected. In September 2024, the review considered seven databases: ACM Digital Library[1], IEEE Digital Library[2], Science@Direct[3], Springer Link[4], Wiley[5], Scopus[6] and MDPI[7]. Previous literature reviews ((ZANELLA GOMES et al., 2019; GUSENBAUER; HADDAWAY, 2020; ARANDA et al., 2023; VIANNA; BARBOSA, 2017; HECKLER; CARVALHO; BARBOSA, 2022)) also utilized these databases. Table 3 summarizes the search process for each of these databases.

Table 2: Search string.

| **Main term** | **String** |
|---|---|
| Performance Metrics | ("performance param*" OR "performance evaluation metrics" OR "monitor* manag*" OR "performance manag*" OR "performance monitor*" OR "performance metric" OR "resource monitor*") |
| Smart Environment | AND ("smart environment" OR "intelligent environment" OR "IoT" OR "internet of things") |
| Monitoring and self-adaptation techniques | AND ("reinforcement learning" OR "self-adaptive" OR "monitor* technic*" OR "monitor* tool*" OR "machine learning" OR "predictive modeling" OR "artificial intelligence" OR "deep learning") |

**Source:** Elaborated by the author

Table 3: Search customization per database

| **Database** | **Search Description** |
|---|---|
| ACM Digital Library | Search considered the expansion “ACM Guide to Computing Literature”. |
| IEEE Digital Library | Search considered all metadata fields and publication types. |
| Science@Direct | Search considered all metadata fields and publication types. |
| Springer Link | Search considered all metadata fields and publication types “Article”, “Conference Paper” and “Conference Proceedings” and excluded articles in the preview. |
| Wiley | Search considered all metadata fields and publication type. |
| Scopus | Search considered the fields article title, abstracts, and key words. |
| MDPI | Search considered all metadata fields and publication type. |

**Source:** Elaborated by the author

[1]https://dl.acm.org/.
[2]https://ieeexplore.ieee.org/Xplore/home.jsp.
[3]https://www.sciencedirect.com/.
[4]https://link.springer.com/.
[5]https://onlinelibrary.wiley.com/.
[6]https://www.scopus.com.
[7]https://www.mdpi.com/.

### 3.1.3 Selection process

After applying the search string, the articles underwent a filtering process, considering the inclusion (IC) and exclusion (EC) criteria. This study included only articles that met all the inclusion criteria. Conversely, articles that met at least one exclusion criterion were discarded. These criteria were defined to eliminate noise in the research, ensuring the relevance of the selected studies. Table 4 presents the inclusion and exclusion criteria.

Table 4: Inclusion and exclusion criteria of the research.

| Criterion | Definition |
|---|---|
| **Inclusion** | |
| IC1 | Studies published in peer-reviewed journals, conferences, or workshops. |
| IC2 | Articles written in English. |
| IC3 | The study must contain the terms defined in the search string. |
| **Exclusion** | |
| EC1 | Duplicate works. |
| EC2 | The study is a literature review or systematic mapping. |
| EC3 | Works that are not aligned with any of the research questions. |
| EC4 | Publications that do not directly address smart environments or similar contexts. |
| EC5 | Studies that do not address the use of techniques and/or tools for monitoring or performance improvement. |
| EC6 | Works that do not provide analysis of experiments related to the impact of monitoring techniques and/or tools on system performance metrics. |

**Source:** Elaborated by the author

The software Parsifal[8] was used to manage the metadata of the selected articles, assisting in conducting the systematic review from protocol registration to results evaluation. The filtering process was divided into several stages, each addressing different exclusion criteria and article metadata, ensuring speed and clarity in the review process.

The selection process involved multiple phases, with the initial filtering returning 7,763 articles after applying the search string. Then, the three-pass reading method proposed by KESHAV (2016) was applied, allowing for a progressive analysis of the articles. The complete filtering resulted in 133 studies selected for full analysis. Figure 2 illustrates the complete flow of the study selection process.

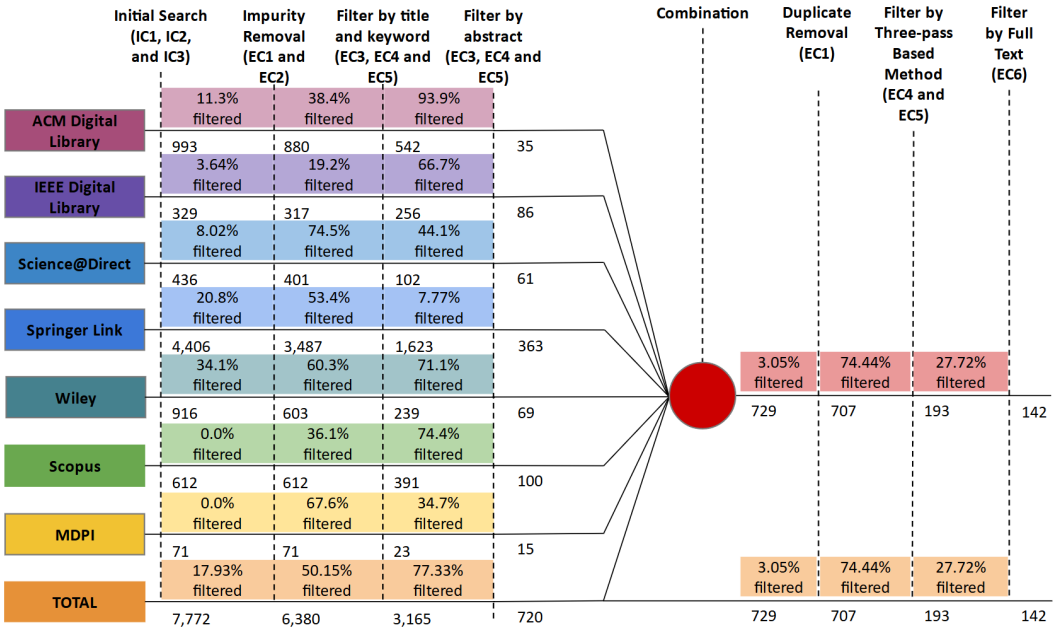
## 3.2 Research Results

Table 5 summarizes the reviewed articles, listing the MAPE (Monitor, Analyze, Plan, Execute) functions addressed by each study, the corresponding references, and the types of Artificial Intelligence (AI) applied. This classification illustrates how monitoring, learning, and self-adaptation techniques are distributed across different contexts and smart environments.

Self-adaptive systems follow the MAPE model to structure their continuous adaptation cycles. The model defines four key functions: Monitoring, responsible for collecting data from the system environment; Analysis, focused on processing the data to identify trends, anomalies, or adaptation needs; Planning, which proposes strategies based on analytical results; and

[8]https://parsif.al

Figure 2: Flow of the study selection process.




**Source:** Elaborated by the author

Execution, where the system applies the planned adaptations. By organizing system behavior into these phases, self-adaptive solutions improve responsiveness and flexibility in dynamic environments (GHEIBI; WEYNS; QUIN, 2021). This structure enables systems to maintain operational goals despite changes in workload, network conditions, or hardware availability, characteristics commonly found in smart and distributed environments.

The reviewed studies reveal a range of AI techniques adopted to implement these functions. Several works explore fuzzy logic to enhance decision-making during the analysis and execution phases, particularly in sensor networks where uncertainties and partial information are common. Fuzzy systems support flexible decision boundaries, making them effective in environments where crisp threshold-based rules perform poorly. In other cases, unsupervised learning methods, such as clustering and anomaly detection algorithms, have been integrated into monitoring architectures to automatically identify behavioral deviations without requiring labeled datasets. These methods enable early detection of anomalies, assisting the planning module in selecting more appropriate responses before major degradations occur.

Reinforcement learning techniques have gained relevance, especially in managing resource allocation and routing in dynamic networked systems. In these approaches, agents learn adaptation strategies through continuous interaction with the system, progressively refining their decisions to optimize performance indicators such as latency, throughput, and energy consumption. Unlike traditional optimization strategies, reinforcement learning allows adaptation policies to evolve over time, offering the flexibility needed in highly variable environments such as wireless sensor networks, edge computing platforms, and IoT deployments.

Research in IoT and edge computing systems often combines supervised and unsupervised learning. Supervised models are typically employed for fault prediction and resource demand estimation, providing a proactive dimension to monitoring processes. These models use historical data to build predictive pipelines capable of signaling potential failures or bottlenecks in advance, supporting faster and more informed planning decisions. Unsupervised learning complements these efforts by continuously analyzing unlabeled operational data to discover new patterns or changes in system behavior that supervised models may not capture due to training limitations.

In addition to isolated applications of learning techniques, some studies propose hybrid approaches that integrate multiple AI models into a unified monitoring and adaptation pipeline. For instance, supervised learning models may trigger anomaly classification, while reinforcement learning agents select and apply corrective actions based on classified states. This hybridization seeks to close the gap between prediction and action, ensuring that monitored insights directly influence system behavior adjustments without human intervention.

Recent publications continue to expand these approaches, applying reinforcement learning to edge resource planning and supervised learning to the monitoring of communication infrastructures. Several works propose architectures where AI models operate across multiple layers, from device-level monitoring to service-level adaptation strategies. These multi-layered architectures aim to handle the increasing complexity of distributed systems by enabling coordinated adaptation decisions, improving overall system stability and resource efficiency.

Table 5: Selected Articles

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A1 | Anomaly Detection | Analyze, Monitor | Unsupervised, Supervised | **GAO et al. (2021)** |
| A2 | Network Latency, QoS | Analyze, Plan | Reinforcement | **LI; ZHANG; LUO (2021)** |
| A3 | Residual Energy from Sensors, Energy Efficiency | Execute, Plan | None | **MORAES et al. (2022)** |
| A4 | Forecast Accuracy, Network Lifespan, SLA and QoS | Analyze, Plan | None | **SHUKLA et al. (2022)** |
| A5 | Energy Efficiency Delay | Analyze, Execute, Plan | Reinforcement | **ULLAH et al. (2022)** |
| A6 | Energy Consumption, Security Data | Analyze, Monitor | Supervised | **LAKHAN et al. (2022)** |
| A7 | Energy Efficiency | Analyze, Monitor | None | **RAO; SAHOO; YANINE (2024)** |
| A8 | Memory Usage, CPU Usage, Processing and Transmission Delay | Analyze, Monitor | Unsupervised | **SAH et al. (2022b)** |
| A9 | Network Latency, Residual Energy from Sensors | Analyze, Monitor, Plan | None | **ISOLANI et al. (2023)** |
| A10 | Network Latency, QoS | Analyze, Plan | Reinforcement | **SOMESULA et al. (2022)** |
| A11 | Memory Usage, CPU Usage | Analyze, Plan | Reinforcement | **CEN; LI (2022b)** |
| A12 | Energy Consumption, Energy Efficiency | Execute, Plan | None | **SUBRAMANIAN et al. (2022)** |
| A13 | Energy Consumption | Analyze, Monitor, Plan | Supervised | **XU et al. (2023)** |
| A14 | Energy Efficiency, Processing and Transmission Delay | Analyze, Plan | None | **AHMED; ABAZEED (2024)** |
| A15 | Performance KPIs, SLA and QoS, Congestion | Analyze, Monitor | None | **DASH; PENG (2022)** |

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A16 | Energy Efficiency, Network Lifespan, Network Latency | Analyze, Monitor | Supervised | **LOGESHWARAN; SHANMUGASUNDARAM; LLORET (2024)** |
| A17 | Anomaly Detection, SLA and QoS, Congestion | Analyze, Execute, Plan | Reinforcement | **YANG et al. (2021b)** |
| A18 | Network Latency | Execute, Plan | None | **SHUKRY; FAHMY (2024)** |
| A19 | Energy Consumption, Network Latency | Analyze, Plan | Reinforcement, Supervised | **PRAMOD KUMAR; SAGAR (2023)** |
| A20 | Processing and Transmission Delay | Analyze, Plan | Reinforcement | **SANGEETHA et al. (2024)** |
| A21 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **BHARGAVA et al. (2022)** |
| A22 | Energy Consumption | Plan | Reinforcement | **SULIMANI et al. (2023)** |
| A23 | SLA and QoS, Congestion | Analyze, Monitor | Unsupervised | **LIU et al. (2021a)** |
| A24 | Memory Usage, CPU Usage | Analyze, Execute, Plan | None | **AGRAWAL (2023)** |
| A25 | Processing and Transmission Delay | Analyze, Plan | None | **REVANESH et al. (2023)** |
| A26 | Energy Efficiency | Analyze, Execute, Plan | None | **SINGH; MALHOTRA (2018)** |
| A27 | Residual Energy from Sensors, Processing and Transmission Delay | Analyze, Plan | Reinforcement | **Carballido Villaverde; REA; PESCH (2012)** |
| A28 | Energy Efficiency, Energy Consumption | Analyze, Plan | Reinforcement | **YOUSEFI et al. (2020)** |
| A29 | Processing and Transmission Delay | Analyze, Plan | Supervised, Reinforcement | **SHIRMARZ; GHAFFARI (2021)** |
| A30 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **CHAKRABARTI; SADHU; PAL (2023)** |
| A31 | SLA and QoS | Analyze, Monitor | Unsupervised, Supervised | **MIRDULA; ROOPA (2023)** |
| A32 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **SINGH et al. (2021)** |
| A33 | Response Time | Analyze, Monitor | Supervised | **CHOULIARAS; SOTIRIADIS (2020)** |
| A34 | Memory Usage, CPU Usage | Analyze, Monitor | Supervised | **BADUGE et al. (2022)** |
| A35 | Response Time, Performance KPIs, Memory Usage, CPU Usage | Analyze, Monitor | None | **BATTULA et al. (2020)** |
| A36 | Power Consumption/Transmission, Energy Consumption | Analyze, Monitor | None | **S; KANNIGA (2023)** |
| A37 | Security Data, Security and Log Monitoring, SLA and QoS | Analyze, Monitor | Supervised | **XUE et al. (2023)** |
| A38 | Processing and Transmission Delay | Plan | Reinforcement | **MADHUNALA; ANANTHA (2022)** |
| A39 | Response Time, Memory Usage, CPU Usage, Energy Consumption | Analyze, Execute, Monitor, Plan | None | **COLOMBO et al. (2022)** |
| A40 | Energy Consumption | Analyze, Monitor, Plan | Reinforcement | **XU; LIU; PAN (2023)** |
| A41 | Processing and Transmission Delay | Analyze, Monitor, Plan | Supervised | **PRASANNA et al. (2023)** |
| A42 | Security Data, SLA and QoS, Forecast Accuracy, Detection Rates | Analyze, Monitor | Unsupervised | **SELIM et al. (2021)** |
| A43 | Processing and Transmission Delay | Analyze, Execute, Monitor, Plan | None | **BALI et al. (2020)** |
| A44 | Error Rates | Analyze, Monitor | Reinforcement | **SHARMA; SINGH (2020)** |
| A45 | Security and Log Monitoring, Security Data, SLA and QoS, Forecast Accuracy, Detection Rates | Analyze, Plan | Unsupervised, Supervised | **ALNAFESSAH; CASALE (2020)** |
| A46 | Processing and Transmission Delay | Analyze, Monitor | Reinforcement | **WEERASINGHE et al. (2024)** |
| A47 | Core Temperature, Network Lifespan, Congestion | Monitor | Unsupervised, Supervised | **STEHLE et al. (2024)** |
| A48 | Processing and Transmission Delay, SLA and QoS | Analyze, Monitor | Supervised | **SATER; HAMZA (2021)** |

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A49 | Response Time, Performance KPIs, Processing and Transmission Delay | Monitor | None | **HOSEINY et al. (2021)** |
| A50 | Processing and Transmission Delay | Analyze, Execute, Monitor, Plan | Reinforcement | **WU et al. (2023)** |
| A51 | Energy Consumption | Monitor | Unsupervised | **HENNING; HASSELBRING (2019)** |
| A52 | Power Consumption/Transmission, Energy Consumption | Analyze, Monitor | Supervised | **SHAFIEE; OZEV (2022)** |
| A53 | Processing and Transmission Delay, Performance KPIs, Energy Consumption | Analyze, Monitor, Plan | Reinforcement, Supervised | **AZARI et al. (2021)** |
| A54 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **WEI et al. (2022)** |
| A55 | Energy Consumption | Analyze, Monitor | Unsupervised, Supervised | **LI et al. (2024a)** |
| A56 | Energy Consumption | Analyze, Execute, Plan | Supervised | **IMRAN; IQBAL; KIM (2022)** |
| A57 | Processing and Transmission Delay, Network Latency, Network Lifespan, Memory Usage, CPU Usage, Energy Consumption | Analyze, Monitor | Reinforcement | **KHIATI; DJENOURI (2018)** |
| A58 | Processing and Transmission Delay, Network Lifespan, Energy Consumption | Analyze, Monitor | Supervised | **MOCNEJ et al. (2018)** |
| A59 | Processing and Transmission Delay | Analyze, Execute, Plan | Unsupervised, Supervised | **ALKANHEL et al. (2024)** |
| A60 | Energy Consumption and Energy Efficiency | Analyze, Monitor, Plan | Reinforcement, Supervised | **SARITHA; SARASVATHI (2024)** |
| A61 | Response Time, Energy Consumption, Network Latency | Analyze, Execute, Plan | Supervised | **PRIYA et al. (2024a)** |
| A62 | Response Time, Memory Usage, CPU Usage | Analyze, Plan | Reinforcement | **ADIL et al. (2024)** |
| A63 | Processing and Transmission Delay, Network Lifespan, Error Rates | Execute, Plan | Reinforcement | **NANDYALA; KIM; CHO (2023)** |
| A64 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **GUPTA; SHARMA (2023)** |
| A65 | Processing and Transmission Delay, Energy Efficiency | Analyze, Monitor | Supervised | **PUTRA et al. (2023)** |
| A66 | Response Time, Processing and Transmission Delay | Analyze, Monitor, Plan | Reinforcement, Supervised | **SAMARAKOON et al. (2023b)** |
| A67 | Response Time, Processing and Transmission Delay, SLA and QoS, Congestion | Execute, Plan | Reinforcement | **TAM; MATH; KIM (2022b)** |
| A68 | Energy Consumption | Analyze, Plan | None | **HUANG et al. (2022)** |
| A69 | Network Latency | Analyze, Monitor | Supervised | **HAMEED et al. (2021a)** |
| A70 | CPU Usage, Network Latency, Energy Consumption | Execute, Plan | None | **MO'TAZ et al. (2021)** |
| A71 | Network Latency | Execute, Plan | Supervised | **SURESHKUMAR; SABENA (2023)** |
| A72 | Response Time, Energy Consumption, Network Latency | Analyze, Plan | Reinforcement | **WU et al. (2021)** |
| A73 | Energy Consumption, Network Latency | Monitor | Supervised | **SUNDARESAN; DURAI (2018)** |
| A74 | Energy Consumption, Network Latency | Analyze, Monitor, Plan | Supervised | **VINJAMURI; RAO (2021)** |
| A75 | Energy Consumption, Network Latency | Analyze, Execute, Plan | None | **STEIN et al. (2020)** |
| A76 | Energy Consumption, Network Latency | Analyze, Plan | Reinforcement | **NAGARAJAN et al. (2023)** |
| A77 | Forecast Accuracy, Detection Rates, Congestion | Analyze, Monitor | Supervised | **LALOTRA et al. (2022)** |
| A78 | Anomaly Detection | Analyze, Execute, Plan | Reinforcement | **VELRAJAN; CERONMANI SHARMILA (2022)** |

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A79 | Processing and Transmission Delay | Analyze, Execute, Plan | Reinforcement, Supervised | **FARAJI-MEHMANDAR; JABBEHDARI; JAVADI (2022)** |
| A80 | Response Time, Processing and Transmission Delay | Analyze, Execute, Plan | Reinforcement, Supervised | **TALAAT (2022)** |
| A81 | SLA and QoS, Congestion | Analyze, Monitor | None | **GONG (2022)** |
| A82 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **MUNISWAMY; VIGNESH (2022)** |
| A83 | Power Consumption/Transmission, Network Lifespan | Execute, Plan | None | **GONG et al. (2022)** |
| A84 | Error Rates, SLA and QoS | Analyze, Plan | Reinforcement | **SORRENTINO; FRANZESE; TRIFIRÒ (2024)** |
| A85 | Response Time, Processing and Transmission Delay | Analyze, Plan | None | **KARUNKUZHALI; MEENAKSHI; LINGAM (2022)** |
| A86 | Security Data, SLA and QoS, Congestion | Analyze, Monitor | Unsupervised, Supervised | **KOHYARNEJADFARD et al. (2022)** |
| A87 | Processing and Transmission Delay, Response Time | Analyze, Execute, Plan | None | **KAUR; ARON (2022)** |
| A88 | Processing and Transmission Delay | Analyze, Plan | None | **DJAMA et al. (2022)** |
| A89 | Residual Energy from Sensors | Execute, Plan | Reinforcement | **SUSAN SHINY; MUTHU KUMAR (2022)** |
| A90 | Processing and Transmission Delay | Analyze, Plan | Reinforcement | **ZHENG et al. (2022)** |
| A91 | Processing and Transmission Delay | Analyze, Monitor, Plan | Supervised | **KHAN et al. (2022)** |
| A92 | Residual Energy from Sensors | Analyze, Monitor, Plan | Unsupervised | **STEPHAN et al. (2021a)** |
| A93 | Response Time | Analyze, Plan | Supervised | **ETEMADI; GHOBAEI-ARANI; SHAHIDINEJAD (2021)** |
| A94 | Processing and Transmission Delay | Analyze, Execute, Plan | None | **RAMKUMAR; VADIVEL (2021)** |
| A95 | Processing and Transmission Delay | Analyze, Plan | Unsupervised, Reinforcement | **JAYARAM; PRABAKARAN (2021)** |
| A96 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **COELHO et al. (2021)** |
| A97 | SLA and QoS, Congestion | Analyze, Execute, Plan | None | **SINGH; CHATURVEDI (2024)** |
| A98 | Processing and Transmission Delay | Analyze, Execute, Plan | Reinforcement, Supervised | **MANFREDI et al. (2022)** |
| A99 | SLA and QoS | Analyze, Execute, Plan | Supervised | **GANESH et al. (2024)** |
| A100 | Network Lifespan | Analyze, Plan | Reinforcement | **XU et al. (2024)** |
| A101 | Network Latency | Analyze, Monitor | Supervised | **PENG; WU (2021)** |
| A102 | Processing and Transmission Delay | Analyze, Monitor | Supervised | **HOU; LU; NAYAK (2023)** |
| A103 | Network Latency, SLA and QoS | Analyze, Monitor, Plan | | **TALAAT et al. (2020)** |
| A104 | Processing and Transmission Delay | Monitor | Supervised | **YOU et al. (2023)** |
| A105 | Memory Usage, CPU Usage | Analyze, Monitor, Plan | Supervised | **ZHOU (2023)** |
| A106 | Processing and Transmission Delay | Analyze, Monitor, Plan | Reinforcement, Supervised | **BEBORTTA et al. (2021)** |
| A107 | Power Consumption/Transmission, Energy Consumption | Analyze, Execute, Plan | None | **WANG; FAN; NIE (2020)** |
| A108 | Response Time, Anomaly Detection | Analyze, Monitor | Unsupervised | **SHARMA; KAUR (2023)** |
| A109 | SLA and QoS, Congestion | Analyze, Monitor | Unsupervised | **MEENA et al. (2023)** |
| A110 | Residual Energy from Sensors, Network Latency, Network Lifespan, Power Consumption/Transmission, Energy Consumption | Analyze, Plan | None | **AZEVEDO ALBUQUERQUE et al. (2023)** |

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A111 | Processing and Transmission Delay, Security Data | Analyze, Execute, Plan | Reinforcement | **SHARMA et al. (2020)** |
| A112 | Processing and Transmission Delay | Monitor | Reinforcement, Supervised | **GOKCESU et al. (2023)** |
| A113 | SLA and QoS, Congestion | Analyze, Monitor | Supervised | **ALIJOYO et al. (2024)** |
| A114 | Processing and Transmission Delay | Analyze, Monitor | Unsupervised, Supervised | **MALEKZADEH (2023)** |
| A115 | SLA and QoS, Congestion, Anomaly Detection, Detection Rates | Analyze, Plan | Reinforcement | **PRABHU; ALAGESWARAN; MIRUNA JOE AMALI (2023)** |
| A116 | Performance KPIs, Security and Log Monitoring, Anomaly Detection | Analyze, Monitor | None | **CALDERON et al. (2023)** |
| A117 | SLA and QoS, Congestion | Analyze, Plan | Reinforcement | **SAKR; ELSABROUTY (2023)** |
| A118 | Processing and Transmission Delay | Analyze, Plan | None | **BACANIN et al. (2024)** |
| A119 | Processing and Transmission Delay | Analyze, Plan | None | **STEPHAN et al. (2021b)** |
| A120 | Processing and Transmission Delay | Analyze, Monitor, Plan | Supervised | **FARAJI-MEHMANDAR; JABBEHDARI; JAVADI (2023)** |
| A121 | Network Lifespan | Analyze, Plan | Reinforcement | **NAZARI et al. (2023)** |
| A122 | Energy Efficiency, Network Lifespan | Analyze, Plan | Reinforcement | **SENNAN et al. (2024)** |
| A123 | Forecast Accuracy, Detection Rates, Anomaly Detection | Analyze, Plan | None | **SOUNDARI; JYOTHI (2020)** |
| A124 | Energy Efficiency, Network Lifespan | Analyze, Plan | Reinforcement | **ESFAHANI; ELKHODARY; MALEK (2013)** |
| A125 | Processing and Transmission Delay, Energy Efficiency | Analyze, Monitor, Plan | Reinforcement, Supervised | **NANDISH; PUSHPARAJESH (2024)** |
| A126 | Response Time, Energy Efficiency, Network Latency | Analyze, Monitor | None | **SRICHANDAN et al. (2024)** |
| A127 | Response Time | Analyze, Plan | Reinforcement | **RATH; MANDAL; SARKAR (2024)** |
| A128 | Energy Consumption, SLA and QoS | Monitor | None | **CALINESCU et al. (2011)** |
| A129 | Network Latency | Execute, Plan | None | **CALINESCU et al. (2020)** |
| A130 | Network Latency, Response Time | Analyze, Plan | Unsupervised | **CHEN et al. (2018)** |
| A131 | Power Consumption/Transmission | Analyze, Plan | Supervised | **CHENG; RAMIREZ; MCKINLEY (2013)** |
| A132 | Processing and Transmission Delay | Analyze, Plan | Supervised | **ELGENDI et al. (2019)** |
| A133 | Memory Usage, CPU Usage, Network Latency | Analyze, Plan | Supervised, Reinforcement | **ELKHODARY; ESFAHANI; MALEK (2012)** |
| A134 | Pattern recognition, Smart home sensor data | Monitor, Analyze, Plan | Unsupervised, Incremental Learning | **HAO; BOUZOUANE; GABOURY (2019)** |
| A135 | Intrusion detection, Energy consumption, Security events | Analyze, Plan | Reinforcement Learning | **JAMSHIDI et al. (2025)** |
| A136 | IoT transactions, Resource usage, Security events | Monitor, Analyze, Plan, Execute | None | **KHAN et al. (2025a)** |
| A137 | Resource usage, Model performance, Energy efficiency | Monitor, Analyze | Supervised | **MATATHAMMAL et al. (2025)** |
| A138 | Environmental data (air quality), Device status | Monitor, Analyze, Plan, Execute | None | **BOMBARDA; RUSCICA; SCANDURRA (2025)** |
| A139 | Satisfaction of Non-Functional Requirements (NFRs), System state | Monitor, Analyze, Plan | Supervised | **GARCIA; SAMIN; BENCOMO (2024)** |

| ID | Monitoring Metrics | MAPE | AI Type | Article |
|---|---|---|---|---|
| A140 | QoS attributes, Service availability | Monitor, Analyze, Plan | Supervised | **VAIDHYANATHAN et al. (2024)** |
| A141 | Intrusion patterns, Network security events | Monitor, Analyze | Supervised | **SOROUR et al. (2025)** |
| A142 | Healthcare events, Energy consumption, Decision-making | Monitor, Analyze, Plan | Evolutionary Algorithms, Game Theory | **SELVARAJAN et al. (2025)** |

**Source:** Elaborated by the author

### 3.2.1 GQ1: How are performance metrics being monitored and systems being self-adapted in studies related to Smart Environments?

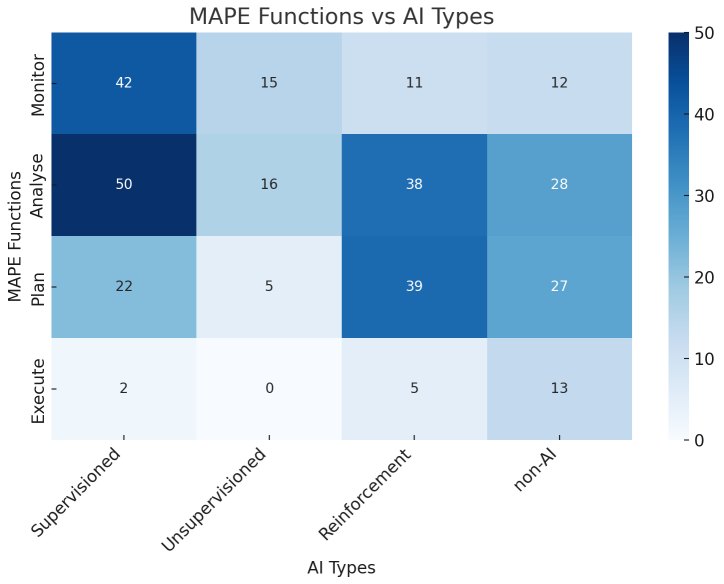
In smart environments, the monitoring of performance metrics supports the evaluation of data integrity and the effectiveness of capture, processing, and management systems (CALINESCU et al., 2011; CHENG; RAMIREZ; MCKINLEY, 2013; RAMKUMAR; VADIVEL, 2021). The reviewed studies commonly monitor indicators such as CPU and memory usage, latency, energy consumption, quality of service, response time, anomaly occurrences, and energy efficiency. These metrics are often supported by IoT infrastructures and distributed systems. They also serve as inputs for self-adaptation mechanisms, particularly those employing AI-based decision-making models.

Figure 4 illustrates the distribution of AI techniques applied in these environments, including Supervised Learning, Reinforcement Learning, and Unsupervised Learning (GAO et al., 2021; SAH et al., 2022b; GUSENBAUER; HADDAWAY, 2020; SOROUR et al., 2025). Supervised Learning appears in 35% of studies, while Reinforcement Learning and non-AI techniques are present in 29% and 24%, respectively. Figure 3 presents a mapping between these AI types and the different stages of the MAPE feedback loop. This loop—composed of Monitoring, Analysis, Planning, and Execution—organizes adaptive processes by identifying which phases are automated to support self-adaptation (RAO; SAHOO; YANINE, 2024; ISOLANI et al., 2023; SHIRMARZ; GHAFFARI, 2021).

In the Monitoring phase, systems collect and analyze real-time metrics to infer current conditions. Supervised Learning models are used to predict performance trends based on historical patterns (GAO et al., 2021; LAKHAN et al., 2022; PRASANNA et al., 2023). Unsupervised Learning techniques support anomaly detection by identifying deviations from regular system behavior (SAH et al., 2022b; SELIM et al., 2021; CHAKRABARTI; SADHU; PAL, 2023). These methods provide a basis for early detection of performance issues.

During the Analysis phase, collected data are evaluated to identify anomalies, performance drops, or optimization possibilities. Neural networks are applied to forecast future conditions, enabling proactive responses (RAO; SAHOO; YANINE, 2024; CHAKRABARTI; SADHU; PAL, 2023). Additionally, clustering algorithms group operational patterns without the need

Figure 3: Mapping of MAPE Functions against AI Types




**Source:** Elaborated by the author

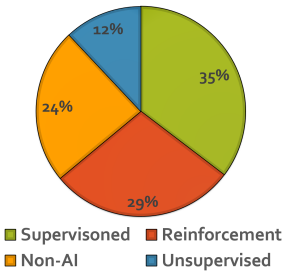
for labeled data, helping to characterize behavior trends and detect irregularities (REVANESH et al., 2023).

The Planning phase defines actions in response to the identified conditions. Reinforcement Learning agents select actions based on system feedback and performance goals (LI; ZHANG; LUO, 2021; GAO et al., 2021; VAIDHYANATHAN et al., 2024). Hybrid approaches combine Supervised Learning for modeling system dynamics and Reinforcement Learning for policy definition, enhancing the planning capability in changing environments (SHIRMARZ; GHAFFARI, 2021).

In the Execution phase, the system implements planned adaptations. While some implementations use AI to refine this step, many still apply rule-based actions triggered by thresholds (HENNING; HASSELBRING, 2019; CHAKRABARTI; SADHU; PAL, 2023). This can limit adaptability when dealing with dynamic workloads.

The integration of AI throughout the MAPE loop enables adaptive behavior aligned with observed and predicted system conditions. Predictive models are used in the Monitoring and Analysis phases to detect performance deviations, while Reinforcement Learning agents guide planning and action selection. Execution is often handled through scripts or automation services (WU et al., 2023; SAH et al., 2022b). Some studies combine AI with rule-based systems to achieve a trade-off between computational overhead and adaptive flexibility (TALAAT, 2022; NAZARI et al., 2023).

Figure 4: Distribution of types of AI applied in smart environments




**Source:** Elaborated by the author

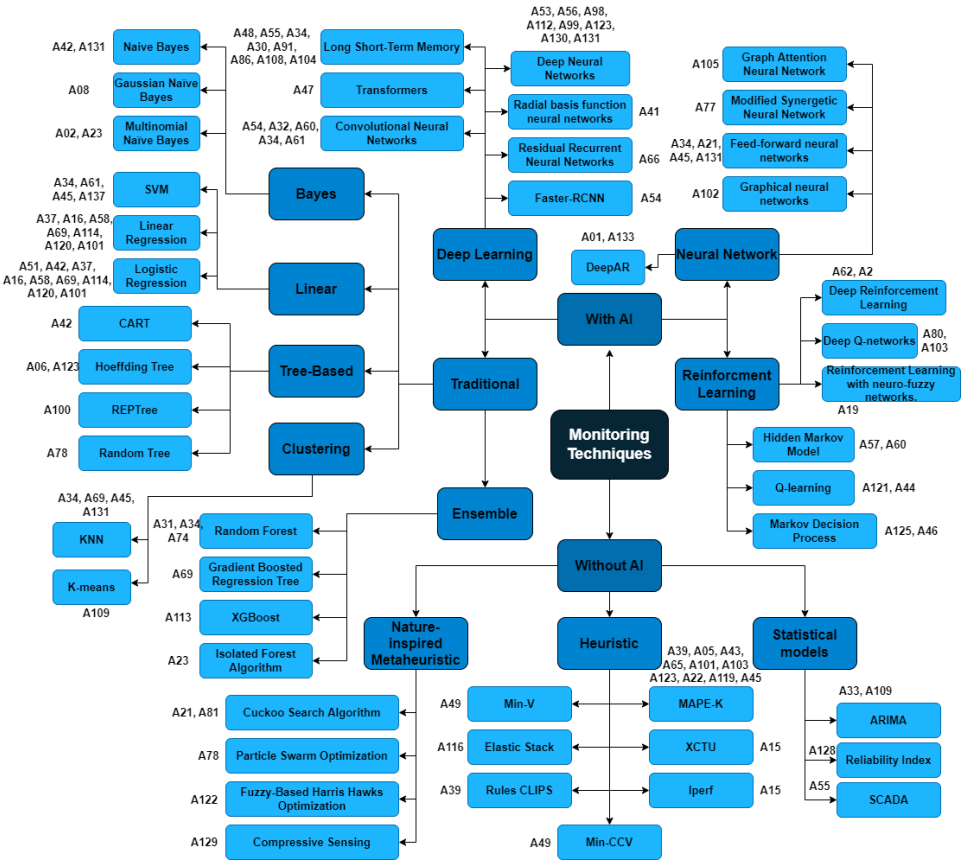
### 3.2.2 FQ1: What techniques and tools are being used to monitor specific performance metrics in smart environments?

To address this question, two taxonomies were developed to categorize the monitoring and self-adaptation techniques identified in the literature. Figure 5 presents the classification of monitoring approaches, while Figure 6 outlines self-adaptation methods. Each taxonomy distinguishes between AI-based and non-AI-based techniques. This classification reflects the diverse strategies employed across systems with varying complexity levels, highlighting how AI methods support enhanced adaptability, while traditional approaches remain relevant for specific scenarios.

Monitoring strategies frequently involve supervised learning models such as Support Vector Machines (SVM), Logistic Regression, and Linear Regression, which are used to evaluate time series data and predict future system states (BADUGE et al., 2022; GAO et al., 2021; XUE et al., 2023). These methods assist in anomaly detection and system stability assessments. Decision tree-based techniques, including CART, Hoeffding Tree, and Random Tree, are adopted to classify recurring behavioral patterns and enable responsive system actions under constrained computational environments (MORAES et al., 2022; LAKHAN et al., 2022). Clustering algorithms like K-means and K-Nearest Neighbors (KNN) support the segmentation of data for identifying abnormal trends, particularly in large-scale deployments (MEENA et al., 2023; HAMEED et al., 2021a). Ensemble learning models such as Random Forest and Gradient Boosted Trees aggregate multiple predictive outputs to enhance detection accuracy in complex datasets (MIRDULA; ROOPA, 2023; CHAKRABARTI; SADHU; PAL, 2023).

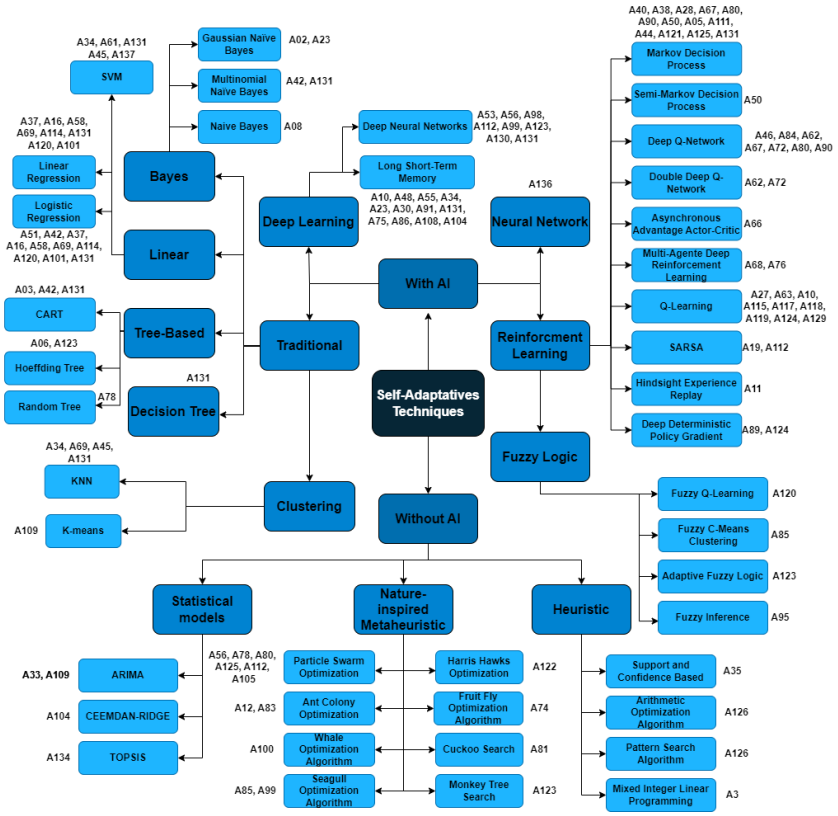
Deep learning techniques have also been applied to performance monitoring tasks. Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks are used to extract features from sensor data and forecast long-term trends, respectively (WEI et al., 2022; GAO et al., 2021). Studies integrate these models to optimize responses to resource load variations and enhance processing performance in edge computing environments (AZARI et al., 2021; MANFREDI et al., 2022). In self-adaptation, reinforcement learning methods

Figure 5: Taxonomy of techniques and tools used for monitoring




**Source:** Elaborated by the author

Figure 6: Taxonomy of techniques and tools used for Self-adaptation




**Source:** Elaborated by the author

such as Q-Learning, SARSA, and Hindsight Experience Replay allow systems to learn configuration adjustments based on interaction with dynamic environments (Carballido Villaverde; REA; PESCH, 2012; SUSAN SHINY; MUTHU KUMAR, 2022). These approaches inform decisions on resource allocation and policy updates without predefined rules.

Nature-inspired algorithms including Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Whale Optimization, and Cuckoo Search are applied to self-adaptation tasks by dynamically tuning parameters and balancing workloads in response to environmental changes (GONG, 2022; SAH et al., 2022b). Complementary to these, statistical models like ARIMA and control systems such as SCADA are used to anticipate failures and manage real-time data (CHOULIARAS; SOTIRIADIS, 2020; LI et al., 2024a). Some studies explore the combination of ARIMA with machine learning to enhance adaptability (RAO; SAHOO; YANINE, 2024). Fuzzy logic-based techniques, such as Fuzzy Q-Learning, Adaptive Fuzzy Logic, and Fuzzy C-Means Clustering, are employed to manage uncertainty and enable adjustments in conditions where deterministic approaches are limited (FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2023; KARUNKUZHALI; MEENAKSHI; LINGAM, 2022; JAYARAM; PRABAKARAN, 2021).

Monitoring and adaptation are also supported by architectural frameworks like MAPE-K

and tools such as the Elastic Stack. These approaches organize the adaptation cycle through continuous monitoring, analysis, planning, and execution of changes based on performance metrics (COLOMBO et al., 2022; CALDERON et al., 2023). MAPE-K has been implemented in sensor networks to guide dynamic adaptations and maintain operational stability (RAO; SAHOO; YANINE, 2024). Hybrid methods that combine neural networks with reinforcement learning are increasingly adopted to improve learning efficiency and resource management under variable operating conditions (SOMESULA et al., 2022; YOUSEFI et al., 2020).

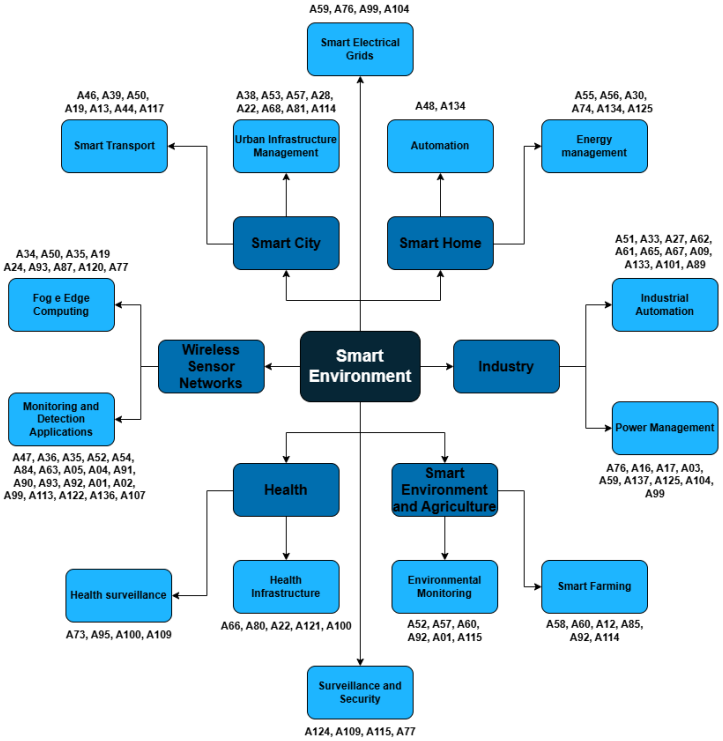
While AI-based methods contribute to adaptive and data-driven decision-making in dynamic environments, their computational requirements and infrastructure demands may not align with all operational contexts. Non-AI techniques, though more limited in flexibility, continue to offer practical solutions for constrained or less variable environments. The studies reviewed indicate that monitoring and adaptation are frequently implemented as separate processes, underscoring the challenge of integrating both dimensions into cohesive and context-aware systems. Selecting appropriate techniques requires considering system goals, resource availability, and the nature of the monitored environment.

### 3.2.3 FQ2: What types of smart environments are covered in the reviewed studies?

Figure 7 illustrates the fourteen main domains of smart environments addressed in the reviewed literature, grouped into six categories. Each environment presents distinct monitoring requirements and adaptation strategies. Smart grids focus on optimizing energy distribution using sensors and control systems, applying machine learning and reinforcement learning to predict consumption patterns and manage infrastructure adjustments (GANESH et al., 2024; ALKANHEL et al., 2024; IMRAN; IQBAL; KIM, 2022; KHIATI; DJENOURI, 2018; ALNAFESSAH; CASALE, 2020). In smart cities, sensor networks and edge computing support the monitoring of traffic, safety, and environmental variables, with machine learning techniques guiding adaptive responses and enhancing urban services (RAO; SAHOO; YANINE, 2024; SAH et al., 2022b; BADUGE et al., 2022; CHAKRABARTI; SADHU; PAL, 2023; PRASANNA et al., 2023).

Smart homes integrate automation platforms to manage comfort and efficiency, relying on predictive analytics to adjust appliance usage based on occupant behavior (CHAKRABARTI; SADHU; PAL, 2023; ALKANHEL et al., 2024; IMRAN; IQBAL; KIM, 2022; NANDISH; PUSHPARAJESH, 2024). Intelligent transportation systems apply machine learning to traffic data and pollution levels, supporting adaptive routing and predictive maintenance with edge computing integration (PRASANNA et al., 2023; BADUGE et al., 2022; GAO et al., 2021; LI; ZHANG; LUO, 2021). In industrial automation, smart sensors and reinforcement learning techniques help monitor equipment conditions and optimize production processes (SUSAN SHINY; MUTHU KUMAR, 2022; SHIRMARZ; GHAFFARI, 2021; CHAKRABARTI; SADHU; PAL, 2023; FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2022).

Figure 7: Taxonomy of smart environments covered




**Source:** Elaborated by the author

Energy management spans residential and industrial contexts, where sensor data and predictive models inform real-time resource allocation (LI et al., 2024a; IMRAN; IQBAL; KIM, 2022; ALNAFESSAH; CASALE, 2020; NANDYALA; KIM; CHO, 2023). Urban infrastructure systems use IoT devices to track utilities such as lighting and water, applying machine learning to predict usage trends and schedule preventive maintenance (RAO; SAHOO; YANINE, 2024; SAH et al., 2022b; CHAKRABARTI; SADHU; PAL, 2023). Fog and edge computing enhance responsiveness by processing data near its source, especially in transportation and health-related systems (GAO et al., 2021; SAH et al., 2022b; CHAKRABARTI; SADHU; PAL, 2023). Wireless sensor networks provide distributed, low-power communication for continuous monitoring across various environments, supporting adaptability in domains like home automation and industrial control (KHIATI; DJENOURI, 2018; SAH et al., 2022b).

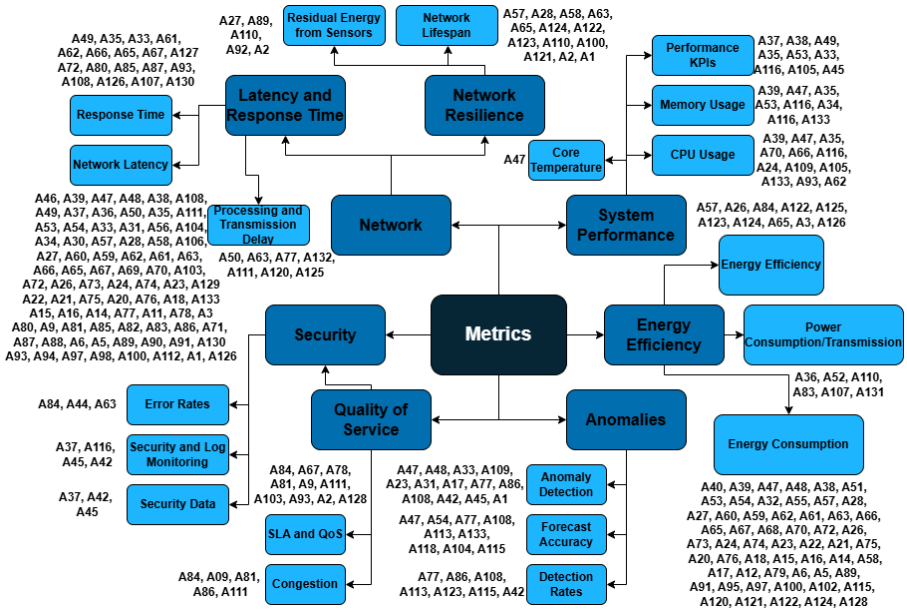
In agriculture and environmental applications, IoT devices monitor soil, climate, and crop health, while predictive models guide irrigation and pest control decisions (STEPHAN et al., 2021a; SINGH et al., 2021; TALAAT, 2022). Health monitoring environments leverage wearable devices and predictive analytics to track patient data and improve response to anomalies (RAO; SAHOO; YANINE, 2024; GAO et al., 2021; IMRAN; IQBAL; KIM, 2022). Lastly, surveillance systems employ AI and anomaly detection models to analyze sensor and video data, adapting surveillance operations and enhancing threat response capabilities (PENG; WU, 2021; SELIM et al., 2021; BALI et al., 2020; SHARMA; SINGH, 2020).

### 3.2.4 FQ3: What specific performance metrics are being monitored in the studies?

The taxonomy in Figure 8 illustrates the variety of performance metrics monitored in the reviewed studies, including latency, energy consumption, resource efficiency, and anomaly detection. Latency and response time are widely evaluated in systems that require real-time processing, such as wireless sensor networks and edge computing scenarios. These studies examine how communication delays affect decision-making and responsiveness, emphasizing the impact of network latency on distributed system behavior (GAO et al., 2021; WU et al., 2023). Resilience is also monitored through metrics like sensor residual energy, network lifespan, and communication stability, particularly in urban IoT infrastructures where maintaining operations during failure or congestion is essential (KHIATI; DJENOURI, 2018; RAO; SAHOO; YANINE, 2024).

Resource usage metrics, including CPU and memory utilization, are commonly tracked to assess processing performance and optimize allocation. Studies highlight how predictive models can anticipate overloads, enabling dynamic scaling and reallocation in fog and edge computing environments (CHAKRABARTI; SADHU; PAL, 2023; IMRAN; IQBAL; KIM, 2022). Energy-related metrics are relevant in industrial and agricultural contexts, where adaptive strategies adjust operational parameters to reduce consumption and align usage with current demands (IMRAN; IQBAL; KIM, 2022; STEPHAN et al., 2021a; RAO; SAHOO; YANINE,

Figure 8: Taxonomy of the metrics monitored




**Source:** Elaborated by the author

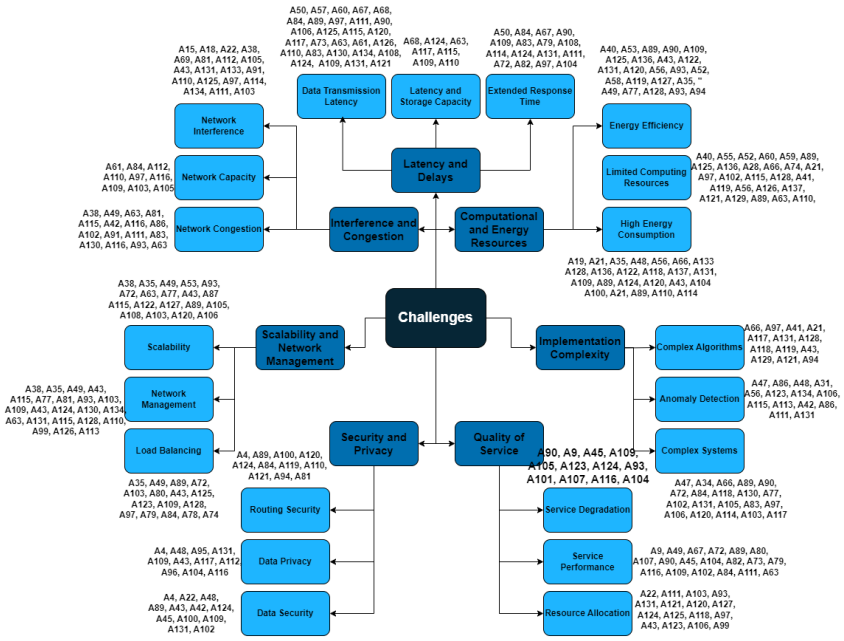
2024). These mechanisms support sustainable operations across varying conditions and load patterns.

Anomaly detection plays a central role in maintaining system stability in critical domains such as surveillance and agriculture. Machine learning models identify behavioral deviations using environmental or infrastructure data, supporting timely interventions to mitigate failures or threats (TALAAT, 2022; SHARMA; SINGH, 2020; MATATHAMMAL et al., 2025). Quality of Service metrics, including error rates, detection of security incidents, and prediction accuracy, are used to evaluate service continuity. Research in this area proposes adaptive policies that adjust network and processing behaviors to preserve reliability during adverse or uncertain conditions (PENG; WU, 2021; FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2022).

### 3.2.5 FQ4: What are the main bottlenecks or challenges identified in the applications of these environments?

The taxonomy in Figure 9 identifies the main bottlenecks in developing self-adaptive systems for smart environments, highlighting areas closely related to key monitored metrics. Latency and response time are frequently cited as constraints in environments that require real-time operation, such as sensor networks and healthcare systems. Studies address this issue by proposing solutions like edge computing, dynamic routing, and traffic prioritization to reduce delays under high device density and variable data transmission demands (GAO et al., 2021; LI; ZHANG; LUO, 2021; RAO; SAHOO; YANINE, 2024).

Figure 9: Taxonomy of the main bottlenecks in creating self-adaptation




**Source:** Elaborated by the author

Another recurring challenge is network congestion and interference, particularly in smart city applications. Dynamic load balancing and congestion-aware routing are proposed as ways to mitigate the effects of growing data traffic and ensure reliable communication (MOCNEJ et al., 2018; GONG, 2022). In parallel, the efficient management of computational and energy resources remains critical, especially in edge and fog computing contexts. Strategies that involve predictive resource allocation and energy-aware scheduling are used to adjust system behavior based on expected workloads (IMRAN; IQBAL; KIM, 2022; STEPHAN et al., 2021a).

As IoT networks scale, coordination and distribution of tasks become more complex. Load balancing techniques and adaptive node management have been proposed to maintain system performance in increasingly distributed infrastructures (ISOLANI et al., 2023; SAH et al., 2022b). However, the integration of machine learning for real-time decision-making introduces additional computational demands. Processing requirements grow as predictive models, anomaly detectors, and decision policies are embedded into runtime architectures (TALAAT, 2022; RAO; SAHOO; YANINE, 2024).

Security and privacy are also identified as relevant bottlenecks in systems dealing with sensitive or distributed data. Research explores the adoption of adaptive security frameworks that combine intrusion detection, data protection, and privacy preservation in dynamic environments (PENG; WU, 2021; ZHOU, 2023). These challenges illustrate the balance that must be achieved between adaptability, performance, and trust in self-adaptive distributed systems. Figure 9 provides a visual synthesis of these issues, supporting the analysis of strategies that address trade-offs among these critical dimensions.

### 3.2.6 FQ5: How are Machine Learning techniques being used to enable self-adaptation in the reviewed systems?

Machine learning techniques support self-adaptation in intelligent systems by enabling them to respond autonomously to environmental changes and operational dynamics. Bayesian models, such as Naive Bayes and Gaussian Naive Bayes, are applied in anomaly detection and classification tasks, allowing the identification of patterns and deviations in distributed sensor data (SELIM et al., 2021; SAH et al., 2022b). Similarly, linear models like Linear and Logistic Regression contribute to forecasting failures and supporting preventive maintenance by correlating historical performance with critical event probabilities (XUE et al., 2023; PENG; WU, 2021).

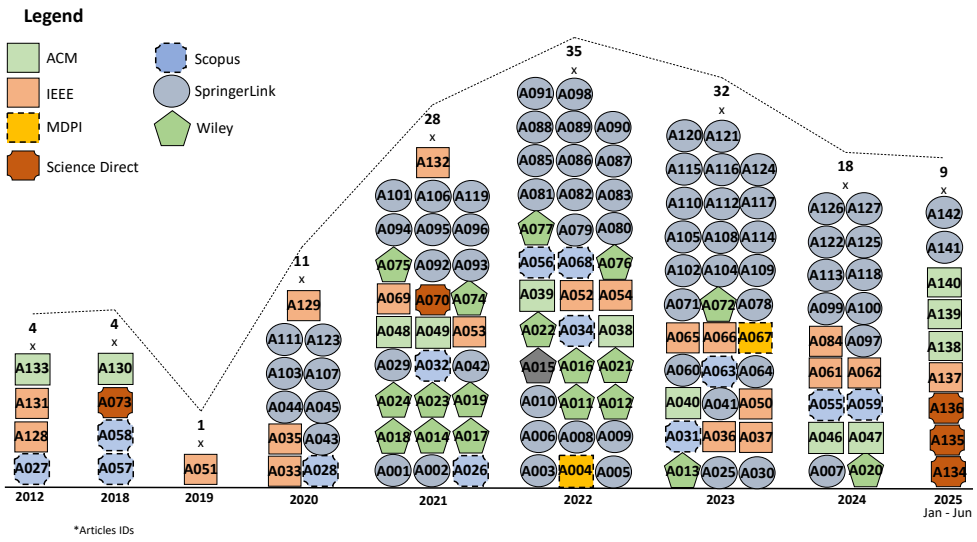
More complex techniques, including deep learning models such as Deep Neural Networks, CNNs, and LSTMs, are employed in predictive monitoring tasks where systems must process continuous sensor data and anticipate behavior trends (AZARI et al., 2021; WEI et al., 2022; SOMESULA et al., 2022). These models enable distributed systems to forecast future conditions and dynamically adjust their configurations. Architectures like Graph Attention Neural Networks and Modified Synergistic Networks are also used to analyze complex relationships and subtle behavioral shifts within high-dimensional data (STEHLE et al., 2024).

Tree-based algorithms, including CART, Hoeffding Tree, REPTree, Random Forest, and Gradient Boosted Trees, offer fast decision-making for real-time anomaly detection and adaptation (SELIM et al., 2021; BADUGE et al., 2022). Unsupervised learning approaches, such as K-means clustering and the Isolated Forest algorithm, address anomaly detection in unlabeled or evolving environments (MEENA et al., 2023). Reinforcement learning methods, including Q-Learning, Deep Q-Networks, and Multi-Agent Deep Reinforcement Learning, are particularly effective in learning adaptive strategies through environmental feedback, enhancing resource allocation and system robustness under uncertain and dynamic conditions (NAZARI et al., 2023; TALAAT, 2022; NANDISH; PUSHPARAJESH, 2024). These techniques contribute to improved resilience and adaptability across smart infrastructures, sensor networks, and industrial systems (GAO et al., 2021; ZHENG et al., 2022).

### 3.2.7 SQ1: Where were the studies published and how has the number of publications evolved per year?

Figure 10 shows the publication sources for the reviewed articles, distributed across ACM, IEEE, MDPI, Science Direct, Wiley, and SpringerLink. These venues encompass conferences and journals focused on self-adaptation in intelligent systems, particularly in areas like sensor networks, distributed systems, and smart cities. In 2021, for example, ACM and IEEE venues hosted a concentration of publications exploring machine learning solutions applied to distributed systems (SATER; HAMZA, 2021; YOUSEFI et al., 2020; YANG et al., 2021b).

Figure 10: Publication Sources




**Source:** Elaborated by the author

The evolution in publication numbers over the years reveals growth in this research area. Publications were relatively low in 2012, with only four articles (ELKHODARY; ESFAHANI; MALEK, 2012; CHENG; RAMIREZ; MCKINLEY, 2013; SELVARAJAN et al., 2025). However, starting in 2020, the number of publications grew exponentially, peaking at 35 in 2022. This increase highlights the rising interest in applying machine learning techniques to self-adaptive systems, as seen in studies like (IMRAN; IQBAL; KIM, 2022; COLOMBO et al., 2022).

In 2023, the number of publications remained high, with 32 articles discussing the integration of supervised and federated learning in complex and data-sensitive scenarios (FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2023; NAZARI et al., 2023; ZHOU, 2023). This trend reflects both the maturation of machine learning techniques and their expanding applicability in diverse domains, including environmental monitoring, healthcare, and network traffic management. In 2024, researchers registered 18 publications, demonstrating the topic's sustained relevance. These studies indicate that combining self-adaptation with AI remains a vibrant area of research and practical application in distributed systems and intelligent environments.

## 3.3 Related Work

This section presents a comparative analysis of prominent self-adaptive architectures identified during the systematic literature review process described earlier in this chapter. The goal is to synthesize how current approaches address system performance, adaptability, and intelligent decision-making within smart environments.

Table 6 provides a structured overview of ten selected architectures that reflect themes relevant to this research, including the use of performance metrics, the presence of feedback loop mechanisms, and the application of artificial intelligence techniques in self-adaptive systems. The selection of these works was guided by objective criteria observed during the literature analysis. Preference was given to studies that present a clear architectural structure focused on runtime adaptation, with an emphasis on the use of performance monitoring data to support decision-making. The presence of formal monitoring and analysis mechanisms, the use of machine learning or nature-inspired algorithms, and the alignment of the study's scope with the goals of this research were also considered. Publication date and citation frequency were used as additional indicators to ensure relevance. These criteria supported the identification of representative contributions that serve as a reference for comparison with the proposed model, highlighting differences in approach, focus, and architectural organization.

The comparison includes the following key performance-related metrics:

- Mean Adaptation Time (MAT) – the average time the system requires to respond to dynamic environmental changes;
- Adaptation Accuracy (AA) – the percentage of adaptation actions that align with intended or expected outcomes;
- Adaptation Overhead (AO) – the additional computational or resource load introduced by the adaptation mechanism;
- Stability – the system's ability to maintain consistent performance following adaptation events.

In addition, the table highlights which phases of the MAPE-K loop (Monitor, Analyze, Plan, Execute, and optionally, Knowledge) are explicitly addressed by each architecture. It also categorizes the types of AI techniques adopted (e.g., supervised learning, unsupervised learning, and reinforcement learning) and identifies which performance metrics are actively monitored by each system to support self-adaptation.

These ten architectures were derived from the broader set of 133 studies retained in the final selection phase of the systematic review, as guided by the methodology proposed by PETERSEN et al. (2008). The inclusion and exclusion criteria ensured that only architectures directly targeting intelligent behavior and dynamic adaptation in IoT or distributed environments were considered. The architectures summarized in Table 6 form the empirical foundation for assessing current trends in performance-driven self-adaptive systems. They offer a representative view of how contemporary approaches incorporate AI-driven decision-making and runtime adaptation to support resilient and efficient smart environments.

SAH et al. (2022a) 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

Table 6: Comparison of self-adaptive architectures

| Architecture | MAT (s) | AA (%) | AO (%) | Stability (%) | MAPE Cycle | AI Type | Metrics Monitored |
|---|---|---|---|---|---|---|---|
| **Oraculum (Proposed)** | 0.05 | 94 | 4 | 98 | M,A,P,E | Supervised, Unsupervised, RL | Parametric (any) |
| (SAH et al., 2022a) | 3.5 | - | 11 | 74 | M,A,E | None | Energy, Traffic Load |
| (VELRAJAN; SHARMILA, 2023) | 1.8 | 94 | - | 79 | M,A,P,E | RL | QoS, SLA |
| (ETEMADI; GHOBAEI-ARANI; SHAHIDINEJAD, 2021) | 1.2 | 91 | - | 84 | M,A | Supervised | CPU, Resources |
| (CEN; LI, 2022a) | 1.15 | 89 | 7 | 87 | A,P | RL | Delay, Resource Allocation |
| (YANG et al., 2021a) | 2.5 | 90 | 10 | 85 | M,A,P | Supervised | Anomaly Detection |
| (LIU et al., 2021b) | 2.7 | 87 | 12 | 82 | A,P | Unsupervised | Anomaly Detection, Data Compression |
| (PRIYA et al., 2024b) | 1.9 | 92 | 7 | 89 | M,A,P | Supervised | IoT Traffic |
| (SAMARAKOON et al., 2023a) | 1.5 | 93 | 6 | 90 | M,A,P,E | None | Latency, Bandwidth, Jitter |
| (HAMEED et al., 2021b) | 2.8 | - | 14 | 75 | M,A,P | Supervised | Throughput |
| (TAM; MATH; KIM, 2022a) | 3.2 | 90 | 13 | 80 | M,A,P,E | RL | Resource, Delay, Priority |
| (KHAN et al., 2025a) | 5.2 | 98 | 12 | 82 | M,A,P,E | Supervised | Resource, Security |
| (SELVARAJAN et al., 2025) | 2.9 | - | - | - | M,A,P | Evolutionary Algorithms, Game Theory | Healthcare events, Energy consumption, Decision-making |

**Source:** Elaborated by the author

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 Adaptation Accuracy (AA), reported an adaptation overhead of 11%, and a network stability of 74%, suggesting limitations under dynamic operating conditions.

VELRAJAN; 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% adaptation accuracy with a mean adaptation time of 1.8 seconds. The study did not assess adaptation overhead, and the reported stability of 79% indicates potential performance variability during migration events.

ETEMADI; GHOBAEI-ARANI; SHAHIDINEJAD (2021) presented an auto-scaling mechanism for IoT applications in fog computing using deep learning for workload-aware resource allocation. The model achieved an adaptation accuracy of 91% with an average adaptation time of 1.2 seconds. However, adaptation overhead was not evaluated, and stability was measured at 84%, suggesting susceptibility to workload fluctuations.

CEN; LI (2022a) 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% adaptation accuracy, 1.15 seconds mean adaptation time, and a 7% adaptation overhead. Stability reached 87%, though the work emphasized

delay minimization without fully addressing other adaptation metrics.

YANG et al. (2021a) 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% adaptation accuracy, 2.5 seconds mean adaptation time, 10% adaptation overhead, and 85% stability. Longer adaptation times and moderate overhead may constrain its real-time deployment potential.

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

PRIYA et al. (2024b) 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% adaptation accuracy, 1.9 seconds adaptation time, 7% overhead, and 89% stability. The study emphasized forecasting rather than runtime adaptation control.

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

HAMEED et al. (2021b) 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 mean adaptation time of 2.8 seconds, adaptation overhead of 14%, and stability of 75%, indicating sensitivity to variable traffic profiles.

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

The Oraculum provides an alternative architecture for self-adaptive systems. Evaluation results report an adaptation accuracy of 94%, mean adaptation time of 0.05 seconds, adaptation overhead of 4%, and system stability of 98%. The framework combines supervised, unsupervised, and RL to support runtime decision-making across various conditions. Its modular structure enables monitoring of discretizable metrics without requiring domain-specific adaptation logic. These results position the architecture as a candidate for deployment in dynamic, heterogeneous IoT environments requiring low-latency adaptation and high resilience.

## 3.4 Considerations about the Chapter

This chapter examined the main monitoring and adaptation strategies used in smart environments, highlighting both AI-based and traditional approaches. Supervised learning techniques were commonly adopted to predict failures, estimate resource usage, and maintain performance in dynamic systems such as sensor networks, energy grids, and healthcare monitoring (GAO et al., 2021; LAKHAN et al., 2022; BADUGE et al., 2022; FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2023; GUPTA; SHARMA, 2023). These methods provided accurate forecasts that improved planning and anomaly detection. Unsupervised learning was also applied in scenarios with high data volumes and limited labels, enabling anomaly detection and system clustering for traffic analysis and energy optimization (SELIM et al., 2021; MEENA et al., 2023; COELHO et al., 2021; SAH et al., 2022b; ISOLANI et al., 2023; RAO; SAHOO; YANINE, 2024).

Reinforcement learning (RL) methods were employed for autonomous decisions in environments with dynamic workloads, supporting tasks like bandwidth management and task allocation (LI; ZHANG; LUO, 2021; PRABHU; ALAGESWARAN; MIRUNA JOE AMALI, 2023; SOMESULA et al., 2022). These techniques frequently relied on Markov Decision Processes (MDPs) to formalize adaptation (XU et al., 2024; NAZARI et al., 2023). However, most RL implementations operated independently, without combining supervised or unsupervised learning (WU et al., 2023; ZHENG et al., 2022). Meanwhile, rule-based, heuristic, and statistical techniques remained prevalent in systems with limited computational resources or strong interpretability requirements (MORAES et al., 2022; SHUKLA et al., 2022; DASH; PENG, 2022), although they lacked adaptability under unpredictable conditions (REVANESH et al., 2023; RAO; SAHOO; YANINE, 2024).

Many works monitored metrics like latency, memory, CPU, and energy (SHUKLA et al., 2022; CHAKRABARTI; SADHU; PAL, 2023), yet few integrated these with structural adaptations such as component replication or reallocation. Hybrid strategies combining AI and traditional mechanisms were underexplored, particularly in edge and fog systems where adaptive behavior must coexist with low overhead (ZHENG et al., 2022; ZHOU, 2023). Although federated learning was proposed to preserve privacy in decentralized scenarios (ZHENG et al., 2022), few studies combined it with other learning paradigms. The reviewed literature also lacked cross-domain frameworks capable of generalization, with most solutions restricted to specific domains such as vehicular networks or energy monitoring.

The proposed taxonomies help mitigate these gaps by organizing adaptation strategies by responsiveness, monitored metrics, and learning paradigm. Integrating prediction and adaptation remains an open challenge: while several systems forecast behavior (GAO et al., 2021; MALEKZADEH, 2023), few connect predictions with proactive adaptation (NAZARI et al., 2023; PRABHU; ALAGESWARAN; MIRUNA JOE AMALI, 2023; CHAKRABARTI; SADHU; PAL, 2023). The taxonomy on adaptation challenges emphasized real-time inference, design

complexity, and privacy as major obstacles (MORAES et al., 2022; MANFREDI et al., 2022). This chapter contributes by categorizing techniques and proposing an architectural foundation that combines prediction, classification, and reinforcement learning across layers, enhanced by an ontology to formalize metric relationships and promote scalable, self-adaptive behavior in smart environments.

# 4 SHIELD SIMULATOR

This chapter introduces the SHiELD (Sensor Heuristics and Intelligent Evaluation for Large-scale Data) computational model, designed for efficient simulation, processing, and forecasting of sensor data within Internet of Things (IoT) environments. The proliferation of IoT technologies has increased the deployment of sensor-based systems across domains such as smart cities, industrial automation, environmental monitoring, and healthcare (REHMAN et al., 2025; MBIMBI; MURRAY; WILSON, 2024; LATHA; John Justin Thangaraj, 2025).

A key challenge involves balancing data transmission efficiency with the quality of retained information. Sensor networks often operate under bandwidth constraints, limited processing capacity, and real-time responsiveness requirements (KIM et al., 2019; ALYMANI et al., 2025). The SHiELD simulator incorporates heuristic-based data processing techniques—aggregation, compression, and filtering—to reduce redundancy, optimize bandwidth use, and decrease processing overhead.

SHiELD integrates time series forecasting mechanisms to anticipate future conditions, supporting pre-trained models like AutoRegressive Integrated Moving Average (ARIMA), effective in capturing temporal patterns in IoT sensor data (LIU et al., 2024; HABBAL; ALI; ABUZARAIDA, 2024). The combination of prediction and heuristic optimization addresses the demand for self-adaptive capabilities in control systems (ZHUANG et al., 2023; GUO et al., 2023). The architecture of SHiELD simulates the entire lifecycle of sensor data: synthetic generation, heuristic processing, predictive modeling, and real-time performance monitoring. Its modular structure allows deployment in both local and distributed environments, supporting varied experimental configurations (ZARE et al., 2024; KORKALAINEN et al., 2009; OSMAN, 2025).

The simulator incorporates concepts from adaptive control and intelligent systems, including event-triggered control, spatiotemporal fault detection, and constrained dynamic programming (WANG et al., 2023; PENG et al., 2024; ZHAO et al., 2023). Unlike earlier platforms focused solely on data generation or prediction, SHiELD integrates simulation, heuristics, and predictive analytics in a unified environment. Subsequent sections detail the conceptual and technical design of the SHiELD simulator, presenting the implemented architecture, heuristic models for data optimization, forecasting strategies, and mechanisms for runtime performance monitoring.

## 4.1 Related Work

The simulation of IoT systems and sensor data processing has been a key area of research, with several simulators developed to support the design, testing, and optimization of IoT applications. Each simulator addresses one of these different aspects of IoT systems: data generation, processing, security, or integration with cloud services.

A systematic search was conducted across six major academic databases to identify rele-

vant studies on sensor data generation, simulation, and predictive modeling: IEEE[1], ACM[2], Springer[3], Scopus[4], ScienceDirect[5], and MDPI[6]. The search was performed using a predefined query string, shown in Table 13, designed to capture research focused on sensor data handling, simulation methodologies, and predictive modeling techniques in IoT environments.

The initial search retrieved a total of 312 studies. The filtering process followed three inclusion criteria: (1) studies published in peer-reviewed journals, conferences, or workshops; (2) articles written in English; and (3) works explicitly discussing sensor data generation, simulation, or predictive modeling. Additionally, the process applied exclusion criteria to remove (1) studies without practical implementation, (2) research unrelated to sensor data processing or simulation, and (3) duplicate records across multiple databases. After applying these criteria, the selection narrowed to 58 articles.

The study refined the selection by applying the three-pass reading method proposed by KESHAV (2016). The first pass reviewed titles and abstracts to identify relevant papers. The second pass examined the introduction and methodology sections to assess alignment with the research scope. Finally, a full-text analysis was conducted in the third pass to ensure that each selected study contributed directly to the research objectives. After this systematic evaluation, a final set of 10 studies was retained, each addressing different aspects of sensor data simulation, processing, or predictive modeling in IoT environments.

Table 7: Definition of the search string for related works

| **Key Terms** | **Search Terms** |
|---|---|
| Sensor Data | ("sensor data" OR "IoT data") AND |
| Simulation | ("simulator" OR "simulation environment" OR "synthetic data generation") AND |
| Predictive Modeling | ("predictive modeling" OR "time-series forecasting" OR "ARIMA") |

**Source:** Elaborated by the author

Several IoT simulators have made significant contributions but share common limitations regarding real-time data processing and predictive modeling. POLLEY et al. (2004) introduced ATEMU, which enables detailed emulation of sensor nodes and supports heterogeneous networks; however, it lacks dynamic data processing and predictive capabilities. Similarly, CHEN et al. (2005) developed SENSE, a simulator optimized for scalability and memory efficiency in large-scale sensor networks, but it does not incorporate real-time analysis or forecasting.

Reviews of IoT simulators and testbeds highlight the need for better integration between virtual and physical domains, as well as the absence of dynamic data handling and predictive modeling in many existing tools (CHERNYSHEV et al., 2018). PFLANZNER et al. (2016)

[1] https://ieeexplore.ieee.org/
[2] https://dl.acm.org/
[3] https://link.springer.com/
[4] https://www.scopus.com/
[5] https://www.sciencedirect.com/
[6] https://www.mdpi.com/

presented MobIoTSim, which allows the simulation of multiple mobile IoT devices and cloud integration, yet it does not support multi-stage data analysis or predictive modeling.

EdgeMiningSim, proposed by SAVAGLIO; FORTINO (2021), enables the development of descriptive and predictive models for sensor data streams, but does not provide real-time processing or multi-stage analysis. In the context of smart waste management, HUSSAIN et al. (2024) introduced a multiagent simulation framework that models sensors and collection routes, though it lacks predictive modeling and dynamic analytics.

IoTSecSim, developed by CHEE et al. (2024), focuses on simulating cyber-attacks and defense mechanisms in IoT environments, but does not offer comprehensive data processing or predictive features. Kaala, introduced by DAYALAN et al. (2022), integrates devices, gateways, and cloud services for end-to-end IoT simulation, yet it is limited by the absence of multi-stage data processing and real-time monitoring.

Finally, NúñEZ; CAñIZARES; de Lara (2022) presented CloudExpert, a system that assists in selecting suitable cloud simulators for various scenarios, which is valuable for IoT-cloud integration research. However, it does not directly address the data processing limitations found in many IoT simulators.

Table 8: Comparison of Sensor Data Simulators and Processing Systems

| Feature | ATEMU | SENSE | Mob IoT-Sim | IoT Sec-Sim | Kaala | Edge Mining Sim | Waste Management | IoT Research | Cloud Expert | SHiELD |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sensor Data Simulation** | Yes | Yes | Yes | No | Yes | Yes | No | Yes | Yes | Yes |
| **Real-time Data Processing** | No | No | No | No | Yes | Yes | No | No | No | Yes |
| **Predictive Modeling (ARIMA)** | No | No | No | No | No | No | No | No | No | Yes |
| **Multi-stage Data Processing** | No | No | No | No | Yes | Yes | No | No | No | Yes |
| **Security Evaluation** | No | No | No | Yes | No | No | No | No | No | Yes |
| **Data Mining Support** | No | No | No | No | No | Yes | No | No | Yes | Yes |
| **Performance Monitoring** | No | No | No | No | Yes | Yes | No | No | Yes | Yes |
| **Customizability** | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

**Source:** Elaborated by the author

Table 8 presents a comparative analysis of the reviewed sensor data simulators and processing systems. Each column represents a different simulator, while the rows indicate specific features, including sensor data simulation, real-time data processing, predictive modeling, multi-stage data processing, and performance monitoring. The comparison highlights the capabilities of SHiELD, which supports the entire lifecycle of sensor data simulation, processing, and real-time monitoring, making it valuable for both research and practical applications in the IoT domain.

Each column in the table represents one of the reviewed simulators: ATEMU (POLLEY et al., 2004), SENSE (CHEN et al., 2005), MobIoTSim (PFLANZNER et al., 2016), IoT-

SecSim (CHEE et al., 2024), Kaala (DAYALAN et al., 2022), EdgeMiningSim (SAVAGLIO; FORTINO, 2021), Waste Management (HUSSAIN et al., 2024), IoT Research (CHERNYSHEV et al., 2018), CloudExpert (NúñEZ; CAñIZARES; de Lara, 2022), and the SHiELD. The rows list features, where "Yes" indicates the presence of a feature in the simulator, and "No" indicates its absence. The first (*Sensor Data Simulation*) row shows whether the simulator can generate synthetic sensor data. The second (*Real-time Data Processing*) row indicates whether the simulator can handle and process sensor data in real time. Predictive Modeling (*ARIMA*) specifies whether the simulator integrates predictive models, such as ARIMA, for forecasting sensor data. The *Multi-stage Data Processing* row indicates whether the simulator supports simulating and processing data through the stages: filtering, aggregation, and compression. Security Evaluation identifies whether the simulator includes features for evaluating cyber-attacks and defense strategies in IoT networks. The *Data Mining Support* row addresses whether the simulator supports tasks like data mining within IoT scenarios. Performance Monitoring highlights whether the simulator includes capabilities for tracking system performance in real time. Lastly, *Customizability* reflects whether the simulator can be adapted to different use cases or customized to meet specific research needs.

Other studies have explored different aspects of IoT simulation. IDRIS; KARUNATHILAKE; FöRSTER (2022) conducted a comparative analysis of LoRa-enabled simulators, evaluating tools such as NS-3, OMNeT++, and LoRaSim in terms of transmission efficiency and resource consumption. While these simulators focus on low-power wide-area network (LPWAN) communication, they do not incorporate data processing heuristics or predictive modeling, limiting their applicability to real-time sensor data optimization. Similarly, HARIS et al. (2019) introduced Sensyml, a simulation environment designed for large-scale IoT applications. Sensyml enables modeling sensor interactions with cloud services but does not include support for real-time processing or predictive analytics (IDRIS; KARUNATHILAKE; FöRSTER, 2022; HARIS et al., 2019).

Other studies emphasize flexible representations of IoT sensors for cloud-based simulation environments. MARKUS; KECSKEMETI; KERTESZ (2017) proposed a model to represent IoT sensors in cloud simulators, facilitating scalability assessments and integration with computational infrastructures. However, this approach lacks specific data optimization techniques and time-series forecasting models, reducing its suitability for applications requiring adaptive decision-making based on historical data. Additionally, ALMUTAIRI; BERGAMI; MORGAN (2024b) presented a comprehensive review of IoT simulators, identifying gaps in support for energy models, security, and scalability. Their findings highlight the need for more versatile simulators capable of handling real-time sensor data processing and predictive modeling. **SHiELD** differentiates itself from these approaches by integrating data processing heuristics, such as aggregation and compression, with predictive models like ARIMA, enabling both sensor data simulation and real-time optimization and analysis (ALMUTAIRI; BERGAMI; MORGAN, 2024b; MARKUS; KECSKEMETI; KERTESZ, 2017).

The reviewed simulators cannot process data in real-time, integrate predictive models like ARIMA, or monitor system performance during simulation. Additionally, they do not offer an integrated solution that spans the entire lifecycle of sensor data processing, from data generation to performance evaluation. While extending existing simulators might seem feasible, real-time processing requires optimized architectures to handle continuous data streams with low latency, predictive modeling introduces computational overhead and necessitates adaptive retraining mechanisms, and performance monitoring demands integrated telemetry and analytics. These are not merely additional features but require architectural modifications to ensure scalability, synchronization, and efficient data handling. This work seeks to address these limitations by providing a simulator that integrates multi-stage data processing, predictive modeling, real-time performance monitoring, and the ability to simulate complex sensor data scenarios. This approach enables researchers and practitioners to simulate, process, and analyze sensor data in an integrated way, making it a useful tool for IoT applications and research.

## 4.2 Methodology

The methodology of this work outlines the development of a simulator for processing sensor data in an IoT environment. The system integrates components responsible for generating synthetic data, processing it through heuristics of aggregation, compression, and filtering, and applying predictive models like ARIMA. A central service manages the data flow, distributing processing and balancing tasks and providing an interface for visualization and analysis of results. The use of real-time messaging enables efficient data exchange between components, supporting performance and prediction accuracy in the IoT context.

### 4.2.1 Architecture

The architecture of the sensor data simulation and processing system consists of components that work together to simulate, process, and analyze IoT sensor data. The SHiELD is divided into two main segments: the Sensor Data Simulator and the Sensor Data Processing Architecture Simulator.

The Sensor Data Simulator, illustrated in Figure 11, generates synthetic sensor data that mimics real-world sensor outputs. This data is crucial for testing and validating IoT systems without requiring physical sensors. Users can configure parameters such as sensor types, data frequencies, and time intervals to control the characteristics of the simulated data. The simulator allows data to be processed using different predictive models, such as ARIMA and stochastic tuning, and provides an interface for users to interact with sensor views, historical data, and statistical analysis components.

The Sensor Data Processing Architecture Simulator, shown in Figure 12, is responsible for handling the generated sensor data, applying processing heuristics, and optimizing data trans-

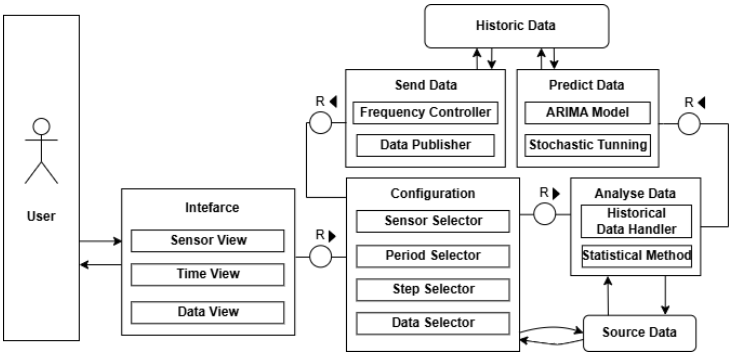
mission. This component consists of an Indoor Server, which captures sensor data and ensures efficient collection through message brokers, and an Outdoor Server, which is responsible for processing the data using balancing mechanisms and performance monitoring services. The architecture also includes a metric collector, allowing system-wide evaluation of real-time performance.

Both figures illustrate complementary aspects of the SHiELD system. Figure 11 focuses on data generation, predictive modeling, and user interaction, while Figure 12 details the system's distributed architecture for processing and optimizing sensor data. Together, they provide a complete representation of how SHiELD enables end-to-end sensor data simulation, processing, and analysis for IoT applications.

Figure 11 shows the structure of the SHiELD. The main components of the simulator include:

- User Interface: This interface allows users to configure simulation parameters, selecting sensor type, frequency, and time intervals. It also provides views for monitoring sensor data in real time, including sensor-specific data, time-based data, and overall data streams.
- Data Publisher: The Data Publisher sends the simulated sensor data to a messaging broker, ensuring the smooth transmission of data to other system components for processing.
- Data Prediction Models: The simulator integrates prediction models like ARIMA and stochastic tuning. These models forecast future sensor values based on historical data, allowing users to test predictive analytics in IoT systems.
- Historical Data Handler: This module processes and stores historical sensor data for analysis and prediction, enabling the system to analyze trends and patterns over time.

Figure 11: SHiELD Architecture

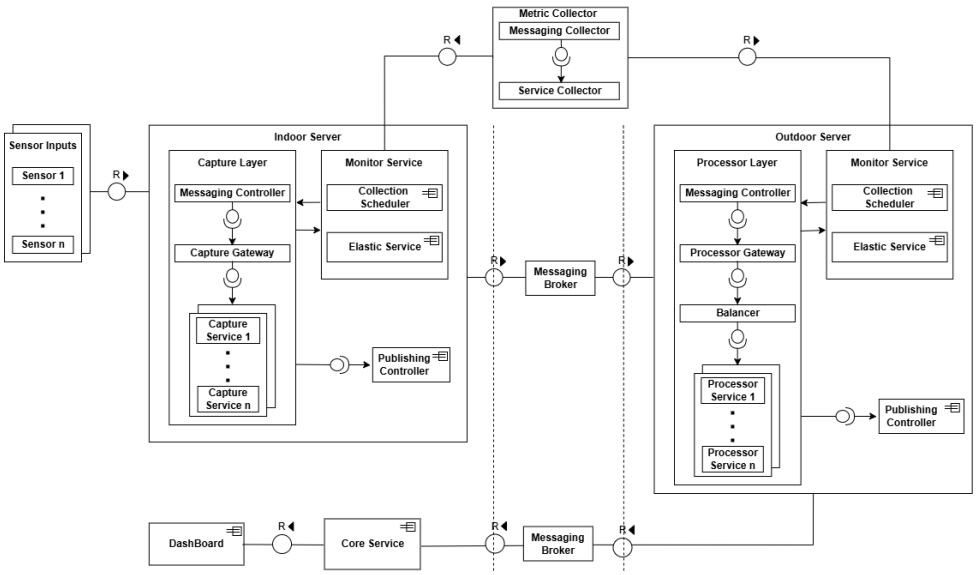



**Source:** Elaborated by the author

The Sensor Data Processing Architecture Simulator models the complete lifecycle of sensor data, from capture to processing and real-time monitoring, operating across a *Local Server* and

an *External Server*. The *Local Server* handles data capture and scheduling. Its Capture Layer manages incoming sensor data via a Messaging Controller and Capture Gateway. Capture Services collect, preprocess, and forward data. A Monitor Service manages collection scheduling and resource elasticity, using a Collection Scheduler and Elastic Services for scaling. The Publishing Controller forwards data to subsequent stages. The *External Server* focuses on processing data. Its Processor Layer connects various processor services for filtering, aggregation, and analyses via a Processor Gateway. A Balancer ensures even load distribution. A Monitor Service schedules collection and manages resource elasticity, with a Publishing Controller to forward processed data.

Figure 12: SHiELD Data Processing Architecture (Local and External Servers)




**Source:** Elaborated by the author

These two components work together to provide a solution for simulating, processing, and analyzing sensor data in an IoT environment. By integrating predictive modeling, real-time data handling, and multi-layer processing, this system can handle the lifecycle of sensor data in various IoT applications, ranging from urban waste management to smart cities and beyond.

### 4.2.2 Prediction Model

The ARIMA (AutoRegressive Integrated Moving Average) model was selected for generating predicted sensor data due to its utility in time series forecasting, particularly for univariate data with trends or seasonality (SURYAWAN et al., 2024). ARIMA consists of autoregression (AR), differencing (I), and moving averages (MA) to capture temporal dependencies. ARIMA was chosen for its ability to handle time-dependent patterns without requiring external variables or complex feature engineering, unlike models such as LSTM. While Exponential Smoothing

(ETS) and basic Moving Average models also use past observations, ARIMA offers a more flexible approach for both stationary and non-stationary data and is computationally efficient for IoT simulations.

The ARIMA model was tested using two publicly available datasets. The first, *Sensor Data*, contains real-world sensor readings (temperature, humidity, pressure) and is suitable for evaluating ARIMA on time series data with seasonal and trend-based components. The second dataset, *Machine Failure Prediction Using Sensor Data*, focuses on machine failure prediction in industrial IoT, allowing for testing ARIMA's ability to predict failure events based on sensor anomalies. These datasets ensure diverse testing scenarios, with the "Sensor Data" dataset serving as a general test case and the "Machine Failure Prediction Using Sensor Data" dataset focusing on predictive tasks. The three main parameters of ARIMA are:

- $p$ (autoregressive order): the number of lag observations included in the model.
- $d$ (degree of differencing): the number of times the data needs to be differenced to achieve stationarity.
- $q$ (moving average order): the size of the moving average window used to smooth residual errors.

The selection of hyperparameters $p = 1$, $d = 1$, and $q = 1$ followed an evaluation of their performance in predicting unseen data during cross-validation (WANG, 2011).

This study adopts the Polygon Area Metric (PAM) as the primary evaluation metric to assess the accuracy of ARIMA's predictions. PAM calculates the area between the predicted and actual values in a time series plot, representing the geometric distance between the two curves. A smaller area indicates better alignment between predictions and actual values. The formula for the *Polygon Area Metric* is:

$$PAM = \frac{1}{2} \sum_{i=1}^{n-1} |(y_i - \hat{y}_i)(t_{i+1} - t_i)| \tag{4.1}$$

where:

- $y_i$ is the actual value at time $i$,
- $\hat{y}_i$ is the predicted value at time $i$,
- $t_i$ is the time step corresponding to the value $i$,
- $n$ is the total number of time steps in the time series.

The Polygon Area Metric (PAM) provides a visual and geometrically intuitive measure of prediction accuracy. Unlike traditional error metrics such as Mean Squared Error (MSE) or Mean Absolute Error (MAE), PAM focuses on the area between the predicted and actual curves,

capturing both large-scale deviations and smaller, localized errors. This is particularly useful in time series forecasting, as it highlights the overall trend and shape of the data rather than individual point discrepancies (AYDEMIR, 2020).

However, PAM alone does not provide a complete measure of prediction accuracy. To supplement its assessment, it is combined with traditional metrics, which offer additional insights into the model's performance. These metrics complement the geometric interpretation of PAM, enabling error analysis and regression performance evaluation. The metrics used in combination with PAM are:

- Accuracy: Measures the proportion of correct predictions made by the model. It is calculated as the ratio of correctly predicted observations to the total observations (WANG, 2011).
- Precision: Measures the proportion of correct identifications. It focuses on the accuracy of positive predictions (WANG, 2011).
- Recall: Also known as sensitivity, this metric measures the proportion of actual positive observations identified. It emphasizes the ability of the model to detect relevant instances.
- F1 Score: The harmonic mean of Precision and Recall, providing a balance between the two. It is useful when the data is imbalanced (ZOLFAGHARI; GHOLAMI, 2021).
- ROC AUC: The area under the receiver operating characteristic curve, which shows the model's ability to distinguish between positive and negative classes. A higher value indicates better performance in distinguishing between the classes (WANG, 2011).

By combining PAM with these traditional metrics, a more thorough evaluation of the ARIMA model's performance is obtained. PAM captures the overall geometric shape of the time series, while the other metrics provide a deeper understanding of the numerical errors, variance, and predictive accuracy. This combination offers a comprehensive framework for assessing time series predictions.

### 4.2.3 Heuristics for Data Processing: Aggregation, Compression, and Filtering

This study developed three heuristics to optimize data transmission: aggregation, compression, and filtering (LI et al., 2024b). These heuristics aim to reduce the volume of transmitted data in IoT systems while preserving essential information. The following are the details of each heuristic and its corresponding algorithm, with additional parameters introduced to enhance the performance and flexibility of each technique (JUAN et al., 2023; ZHANG et al., 2024).

The aggregation heuristic reduces data size by grouping data points into blocks and computing a representative value for each block, typically the average. This method helps compress time-series data without losing significant information. The aggregation algorithm (Algorithm 1) divides the total number of data points by a predefined *blockSize* (line 1) and computes

the average of values within each block. The formula for the aggregated value is expressed as follows:

$$\text{Aggregated Value}_i = \frac{1}{n} \sum_{j=1}^{n} x_{i,j}$$

where $x_{i,j}$ are the values in block $i$ and $n$ is the number of values in the block. One enhancement to this algorithm is the introduction of a *threshold* (Algorithm 1, line 5), which excludes blocks with excessive variation between data points. Additionally, a *weights* mechanism (Algorithm 1, line 1) can be applied to assign different significance to values within each block. For the aggregation heuristic (Algorithm 1), the new length of the aggregated matrix is calculated using *newLength* (line 1), which divides the total length of the original matrix by *blockSize*. The algorithm then sums the values within each block and stores the average in the aggregated result.

**Algorithm 1** Data Aggregation

**Require:** Sensor data array *elements*, array length *length*, block size *blockSize*, threshold *threshold*
**Ensure:** Aggregated data array *aggregated*
1: $newLength \leftarrow \left\lceil \frac{length+blockSize-1}{blockSize} \right\rceil$
2: **for** $i = 0$ to $newLength - 1$ **do**
3: $sum \leftarrow 0$
4: $count \leftarrow 0$
5: **for** $j = 0$ to $blockSize - 1$ **and** $(i \times blockSize + j) < length$ **do**
6: $curr \leftarrow elements[i \times blockSize + j]$
7: $prev \leftarrow elements[i \times blockSize + j - 1]$
8: **if** $|curr - prev| < threshold$ **then**
9: $sum \leftarrow sum + curr$
10: $count \leftarrow count + 1$
11: **end if**
12: **end for**
13: **if** $count > 0$ **then**
14: $aggregated[i] \leftarrow \frac{sum}{count}$
15: **else**
16: $aggregated[i] \leftarrow elements[i \times blockSize]$ {Use original if no valid aggregation}
17: **end if**
18: **end for**
19: **return** $aggregated$

**Source:** Elaborated by the author

The *threshold* (Algorithm 1, line 5) prevents the aggregation of data points with large variations, avoiding distortions in the summarized output. The inclusion of *weights* (Algorithm 1, line 1) provides flexibility in the aggregation process, allowing greater significance for specific data points within a block, making the approach more adaptable to IoT scenarios.

The compression heuristic aims to reduce data size by removing redundant values. Specifically, it eliminates consecutive repeated values, which is useful for time-series data where many consecutive measurements are identical or show minimal variation. By adjusting the threshold for acceptable variation between consecutive values, the algorithm becomes more selective in

compressing the data. Additionally, the compression ratio parameter can be adjusted to control the level of compression applied, depending on the amount of redundancy in the data. The formula for compression is:

$$\text{Compressed Value}_i = \begin{cases} x_i & \text{if } |x_i - x_{i-1}| > \text{threshold} \\ \text{skip} & \text{otherwise} \end{cases}$$

For the compression heuristic (Algorithm 2), the algorithm compares each value with the previous one. When the difference exceeds the threshold, it adds the current value to the compressed array.

**Algorithm 2** Data Compression

**Require:** Sensor data array *elements*, array length *length*, threshold *threshold*, compression ratio *compressionRatio*
**Ensure:** Compressed data array *compressed*
1: $compressedIndex \leftarrow 0$
2: **for** $i = 0$ to $length - 1$ **do**
3: **if** $i = 0$ **or** $|elements[i] - elements[i-1]| > threshold$ **then**
4: $compressed[compressedIndex] \leftarrow elements[i]$
5: $compressedIndex \leftarrow compressedIndex + 1$
6: **end if**
7: **end for**
8: **if** $compressedIndex > length \times compressionRatio$ **then**
9: **print** "Compression is too aggressive, adjust compressionRatio."
10: **end if**
11: **return** $compressed$

**Source:** Elaborated by the author

The *threshold* (Algorithm 2, line 3) controls how sensitive the algorithm is to changes in the sensor data. A lower value makes smaller fluctuations significant, reducing compression, while a higher value allows minor variations to be ignored, further reducing dataset size.

The filtering heuristic reduces noise in sensor data by smoothing rapid fluctuations. This process applies a moving average filter, replacing each data point with the average of its neighboring points within a defined *windowSize* (Algorithm 3, line 1). The *tolerance* (Algorithm 3, line 3) parameter determines the level of smoothing by adjusting the influence of distant points.

For the filtering heuristic (Algorithm 3), the process iterates over the sensor data array and computes the average of the current data point and its neighbors. The *windowSize* (Algorithm 3, line 1) defines how many neighboring points contribute to the smoothing process, while the *tolerance* (Algorithm 3, line 3) parameter adjusts the smoothing intensity based on noise levels in the data. The formula for the moving average filter is:

$$\text{FV} = \frac{1}{\text{wSum}} \sum_{j=-windowSize}^{windowSize} \left( \text{e}[i+j] \times \exp\left( \frac{-|j|}{tolerance} \right) \right)$$

where FV represents the filtered value at index $i$, obtained from a weighted sum of neighboring values. The term $\text{e}[i+j]$ corresponds to the sensor data at position $i+j$, including the current

value and its neighbors. The summation runs over a predefined *windowSize* (Algorithm 3, line 1), determining how many neighboring points contribute to the smoothing process.

The weight of each neighboring value follows the exponential function $\exp\left(\frac{-|j|}{tolerance}\right)$, ensuring that closer values have a stronger influence while distant values contribute less. The decay rate is controlled by the *tolerance* (Algorithm 3, line 3) parameter; smaller values lead to a faster decay, emphasizing nearby points, while larger values create a broader smoothing effect (CHHABRA et al., 2022).

The final result maintains the original scale of the data by normalizing with *wSum* (Algorithm 3, line 9), which represents the sum of all applied weights:

$$\text{wSum} = \sum_{j=-windowSize}^{windowSize} \exp\left(\frac{-|j|}{tolerance}\right)$$

This normalization ensures that the filtered value remains consistent with the original data distribution. By adjusting *windowSize* (Algorithm 3, line 1) and *tolerance* (Algorithm 3, line 3), this filtering approach effectively reduces noise while preserving relevant signal patterns in IoT sensor data.

**Algorithm 3** Data Filtering

**Require:** Sensor data array *elements*, array length *length*, window size *windowSize*, calibration tolerance *tolerance*
**Ensure:** Filtered data array *filtered*

1: **for** $i = windowSize$ to $length - windowSize$ **do**
2: $windowSum \leftarrow 0$
3: $weightSum \leftarrow 0$
4: **for** $j = -windowSize$ to $windowSize$ **do**
5: $weight \leftarrow \exp\left(-\frac{|\,j\,|}{tolerance}\right)$
6: $windowSum \leftarrow windowSum + elements[i + j] \times weight$
7: $weightSum \leftarrow weightSum + weight$
8: **end for**
9: $filtered[i] \leftarrow \frac{windowSum}{weightSum}$
10: **end for**
11: **return** $filtered$

**Source:** Elaborated by the author

In this case, *windowSize* (Algorithm 3, line 1) determines how many neighboring values influence the smoothing process, while *tolerance* (Algorithm 3, line 3) provides flexibility in filtering based on the level of noise present in the data. This flexibility is particularly useful for IoT data, where noise levels can vary depending on environmental conditions or sensor quality. The performance of these heuristics is influenced by the choice of parameters:

- *blockSize* (Algorithm 1, line 1) in aggregation determines the amount of data condensed in each block. Larger values lead to greater data reduction, but very large block sizes may obscure short-term variations in the data.

- *windowSize* (Algorithm 3, line 1) in filtering controls the extent of smoothing applied to the data. Larger values smooth the data more, which may reduce noise but could also lessen the system's responsiveness to sudden changes.

- Compression operates without explicit parameters but can be adjusted based on patterns or noise types in the data. The *compressionRatio* parameter (Algorithm 2, line 5) helps manage the extent of compression applied to the data.

Tuning these parameters is important for optimizing data transmission efficiency while ensuring that key information is preserved. The optimal values for each parameter depend on the specific characteristics of the sensor data and the needs of the IoT system. This tuning process can be carried out through empirical testing, where multiple configurations are evaluated to determine their impact on data reduction and model accuracy, or by applying data-driven strategies that adjust parameters based on statistical properties of the input, such as variance, frequency of change, or entropy.

### 4.2.4 Data Flow in the System

This section describes the data flow within the simulator, which ensures efficient data handling and prediction generation in an IoT environment. The process begins with the Sensor Simulator, which generates sensor data such as temperature, humidity, and pressure. This data is continuously sent to the core service, where the data flow at a given time $i$ is determined by the sensor data generated at that moment.

After reaching the core service, the data is routed to Processing Services (P-services) for operations such as filtering, aggregation, and compression. The transformed data at time $i$ depends on the initial data flow and the specific operation applied. In the case of aggregation, the aggregated data at time $i$ is calculated as the average of the data points within a window of $n$ data points, as shown below:

$$\text{Aggregated Data}_i = \frac{1}{n} \sum_{j=i-n+1}^{i} \text{Data Flow}_j$$

Following processing, the data is sent to the scheduler, where it is organized according to priority and system requirements.

The data is then collected, stored, and used for analysis and prediction. For time series forecasting, the ARIMA model is applied, which is represented by the following equation:

$$y_t = c + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

where $y_t$ is the predicted value at time $t$, $\phi_i$ are the autoregressive parameters, $\theta_j$ are the moving average parameters, $\epsilon_t$ is the error term at time $t$, and $c$ is a constant.

Finally, the processed data and predictions are visualized through the managing interface. The entire data flow is managed by an MQTT broker, which coordinates the transfer of data packets between system components.

### 4.2.5 Testing Methodology

The testing was carried out in both local and external environments, using a set of performance metrics to assess the capabilities and limitations of the simulator. The tests took place in two distinct settings: one leveraging a Raspberry Pi 2 for local simulations and the other utilizing more powerful hardware for external scenarios. For both environments, the testing process followed a stress testing methodology, subjecting the system to increasing loads in a controlled manner. The tests used real data instead of synthetic data, selecting two publicly available datasets for their relevance to sensor data processing and machine failure prediction. These datasets include:

- Sensor data[7], which includes readings from types of sensors such as temperature, humidity, and pressure, providing a broad dataset to evaluate the performance of the ARIMA model in general sensor data with seasonal and trend-based components (BYUN, 2019).
- Machine Failure Prediction Using Sensor Data[8], which captures sensor readings indicating potential machine failures, enabling the evaluation of ARIMA's capability to predict anomalies and impending failure events (RTX, 2024).

The tests progressively increased the number of requests processed by the system, running for a total of 2000 seconds to ensure that all processing was completed within this time frame. The tests allowed to record the following key metrics:

- CPU Usage: The percentage of total CPU resources used by each service, measured using Docker and Kubernetes metrics.
- Memory Usage: The amount of memory allocated and used by each service, recorded in megabytes (MB).
- Response Time: The time taken for a service to process and respond to a request, measured in milliseconds (ms).
- Throughput: The number of requests processed per second, recorded in requests per second.
- Error Rate: The percentage of requests that resulted in errors or failures during processing.

[7]Available at: `https://www.kaggle.com/datasets/yungbyun/sensor-data`

[8]Available at: `https://www.kaggle.com/datasets/umerrtx/machine-failure-prediction-using-sensor-data`

To validate the accuracy of ARIMA-based predictions, the forecasted time series were compared to the original dataset values to assess the similarity between predicted and actual sensor behavior. This comparison helped determine the effectiveness of the forecasting model in reproducing real-world dynamics and its applicability to anomaly detection and adaptive decision-making in IoT environments.

The local testing setup involved a Raspberry Pi 2 with the following specifications: Processor: ARM Cortex-A7 (4 cores, 900 MHz); RAM: 1 GB DDR2; Storage: 16 GB microSD card; Operating System: Raspbian OS (based on Debian); Docker version: 20.10.7; Kubernetes version: 1.21.0. The external setup used more powerful hardware with the following specifications: Processor: Intel Core i7-9700K (8 cores, 3.60 GHz); RAM: 16 GB DDR4; Storage: 500 GB SSD; Operating System: Ubuntu 20.04 LTS; Docker version: 20.10.7; Kubernetes version: 1.21.0.

The local and external tests followed the same general testing methodology. They leveraged Docker containers orchestrated by Kubernetes to manage scaling, load balancing, and resource allocation for each service. The analysis of test results, including CPU usage, memory consumption, and throughput, evaluated the performance of both local and external systems under increasing loads. The collected data enabled an assessment of how the system handled varying resource demands, with particular focus on performance during peak usage scenarios, where CPU and memory usage reached higher levels.

The evaluation included Error Rate as an additional metric to track the reliability of services under stress. This metric ensured that the analysis considered not only throughput and resource consumption but also the accuracy and robustness of the system under heavy load. The methodology focused on realistic IoT environments, with specific focus on how the system scales and handles both normal and peak load conditions, while providing insights into potential bottlenecks and areas for improvement.

## 4.3 SHiELD Results

This section presents the results of the experiments conducted to evaluate the performance of the developed simulator, focusing on aspects such as system architecture, data processing heuristics, and prediction accuracy. The evaluation includes tests in both local and external environments, using a combination of synthetic and real-world sensor data (BYUN, 2019; RTX, 2024; RAJ; HEMA, 2025).

The analysis starts with an evaluation of the system's architectural performance, highlighting its scalability under different load conditions in both environments. The subsequent subsections explore the application of data processing heuristics, namely, aggregation, compression, and filtering, highlighting their impact on the efficiency and size of transmitted data. Finally, we evaluate the ARIMA model's predictive performance using a range of metrics, demonstrating its ability to forecast sensor data trends. These results aim to provide an overview of how the

system performs in different metrics and environments, offering insights into its potential for real-world IoT applications. The application is available in the GitHub repository [9]

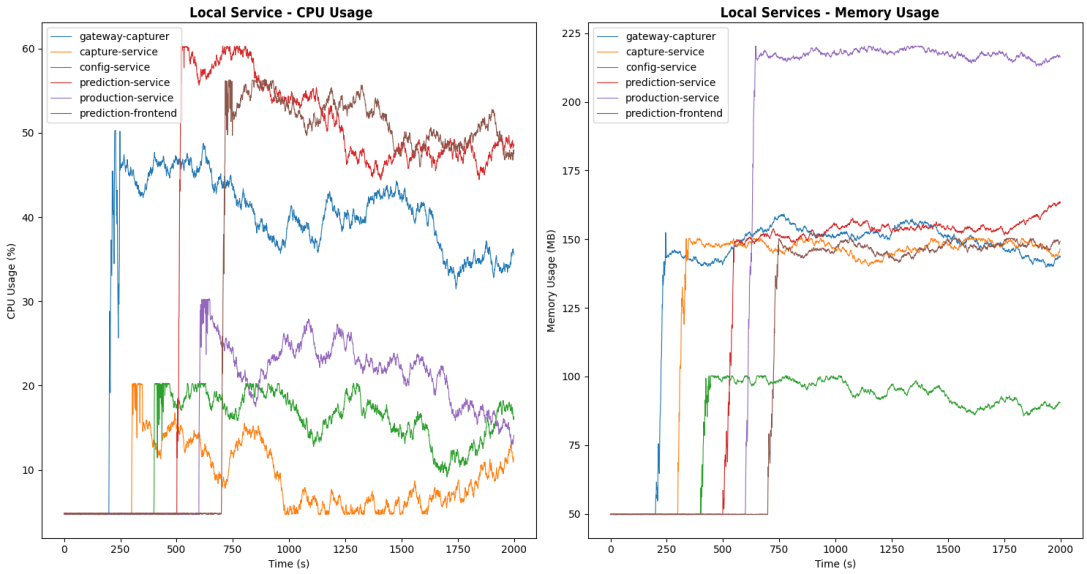
### 4.3.1 Local System Architecture Results

The local system, tested on a Raspberry Pi 2, exhibited CPU usage fluctuations between 10% and 60%. Figure 13 shows noticeable spikes, particularly during intensive tasks executed by the `prediction-service`. These spikes indicate the system's adjustment to increasing load during the tests. The smooth transitions in CPU usage suggest a controlled response to the workload, with occasional minor fluctuations due to varying system demands.

Memory usage remained within the 1 GB capacity of the Raspberry Pi 2 throughout the test. Figure 13 shows periodic spikes, which became more evident when CPU usage increased. These spikes occurred most notably in the `prediction-service` during high-load intervals. Despite these increases, the system stayed within the available memory, demonstrating that the Raspberry Pi 2's resources were sufficient to handle the required tasks during the testing phase.

The results show that the Raspberry Pi 2 was able to manage the stress tests effectively, with memory and CPU usage staying within the hardware's limitations, even during periods of high demand. These fluctuations in CPU and memory usage reflect the system's scalability and its ability to adjust to varying workloads.

Figure 13: Local Services - CPU and Memory Usage




**Source:** Elaborated by the author

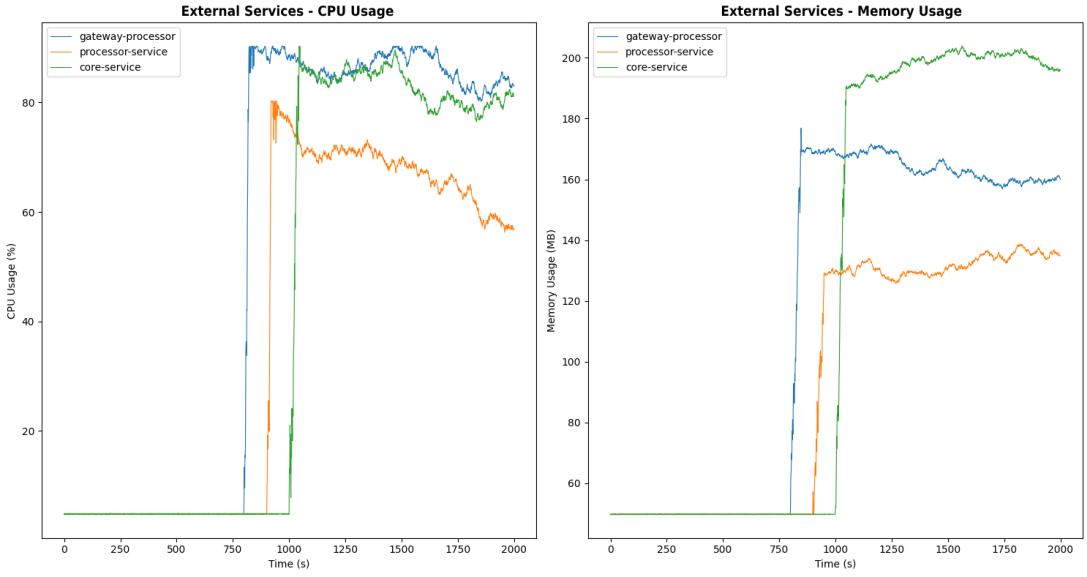
[9] `https://github.com/DarlanNoetzold/SensorSimulator`.

### 4.3.2 External System Architecture Results

The external system, running on more powerful hardware, exhibited a different performance profile compared to the local system. As illustrated in Figure 14, CPU usage fluctuated between 40% and 90%, with noticeable spikes during computationally demanding tasks, particularly those managed by the `processor-service`. The `gateway-processor` maintained relatively lower CPU usage, indicating its ability to handle workloads with less computational overhead than processor-intensive services. These high CPU spikes, especially in the `processor-service`, correspond to moments of complex computations that required increased processing resources.

Memory usage in the external system ranged between 100 MB and 200 MB. Spikes appeared during peak load times, with the `processor-service` experiencing a notable increase in memory demand. This behavior is typical for services that process large datasets or perform complex computations. Despite these spikes, the system efficiently managed the larger data volume, staying within the hardware's memory limits and maintaining performance throughout the test. The system handled the increased workload common in external environments without overburdening the available resources.

Figure 14: External Services - CPU and Memory Usage




**Source:** Elaborated by the author

### 4.3.3 Summary of Architecture Results

Table 9 summarizes the average CPU usage, memory usage, response time, throughput, and error rate for both local and external services during stress testing. The external services,

designed to process larger amounts of data and handle more complex tasks, exhibited higher CPU and memory usage compared to local services.

Table 9: System Architecture Testing Results

| Service | CPU Usage (%) | Memory Usage (MB) | Response Time (ms) | Throughput (requests/sec) | Error Rate (%) |
|---|---|---|---|---|---|
| `prediction-frontend` | 15.3 | 150.5 | 119.6 | 149.7 | 0.5 |
| `config-service` | 10.2 | 110.3 | 79.8 | 199.8 | 0.2 |
| `prediction-service` | 19.7 | 220.8 | 200.3 | 119.8 | 1.2 |
| `production-service` | 30.4 | 180.6 | 150.2 | 181.5 | 0.8 |
| `gateway-capturer` | 24.8 | 190.2 | 171.4 | 160.2 | 0.6 |
| `capture-service` | 40.3 | 310.7 | 220.6 | 140.4 | 1.3 |
| `gateway-processor` | 35.1 | 502.3 | 211.7 | 131.0 | 0.7 |
| `processor-service` | 60.4 | 870.1 | 300.2 | 110.5 | 1.9 |
| `core-service` | 17.8 | 295.6 | 249.9 | 99.7 | 0.4 |
| `metrics-dashboard` | 11.9 | 198.9 | 110.3 | 189.6 | 0.2 |

**Source:** Elaborated by the author

The CPU usage for external services ranged from 40% to 90%, with noticeable spikes observed in processor-intensive services such as `processor-service`, which showed CPU usage peaking around 60%. Memory usage for external services varied between 100 MB and 350 MB during peak periods, with the `processor-service` using the most memory. In comparison, local services showed lower memory consumption, with the peak memory usage for `prediction-service` reaching a maximum of 220 MB. The CPU usage for local services stayed between 10% and 50%.

Both systems performed well under stress testing, with memory usage remaining within the hardware limits for both setups. These results suggest that the SHiELD architecture can scale effectively, with external services utilizing higher available resources to manage larger data volumes and more complex tasks.

### 4.3.4 Heuristics for Data Processing: Aggregation, Compression, and Filtering Results

This section presents the results of applying the aggregation, compression, and filtering heuristics to sensor data. The table below displays the data size after applying compression and filtering heuristics, while the aggregation effect on the number of packets will be discussed separately.

Table 10 presents the effects of compression and filtering on data size. Compression reduces the data size by approximately 5-7% in different load cases, and filtering further reduces the data by 5-6% on average. However, aggregation does not affect the size but reduces the number of packets. The results show how compression and filtering reduce the size of the transmitted data, improving bandwidth efficiency.

Aggregation works by grouping data into larger packets, which reduces the total number of packets sent. This table illustrates the effect of aggregation on the number of packets and the

Table 10: Total Data Volume Before and After Compression and Filtering (KB)

| Data Type | Before Heuristics (KB) | Data after Compression (KB) | Data after Filtering (KB) |
|---|---|---|---|
| Sensor Data (Low Load) | 5300.0 | 4795.3 | 4585.3 |
| Sensor Data (Medium Load) | 8500.0 | 7785.2 | 7461.9 |
| Sensor Data (High Load) | 10700.0 | 9880.9 | 9404.7 |
| Sensor Data (Peak Load) | 15000.0 | 14435.2 | 13710.3 |
| Sensor Data (Normal Load) | 7100.0 | 6840.0 | 6480.0 |

**Source:** Elaborated by the author

data transmitted, calculated as an average per minute for all nodes of the processor-services.

Table 11: Impact of Aggregation on Number of Packets and Data Volume (KB/Minute)

| Data Type | Before Aggregation (Packets/Minute) | After Aggregation (Packets/Minute) | Average Data Transmitted (KB) |
|---|---|---|---|
| Sensor Data (Low Load) | 120 | 20 | 4585.3 |
| Sensor Data (Medium Load) | 200 | 35 | 7461.9 |
| Sensor Data (High Load) | 300 | 55 | 9404.7 |
| Sensor Data (Peak Load) | 400 | 70 | 13710.3 |
| Sensor Data (Normal Load) | 160 | 28 | 6480.0 |

**Source:** Elaborated by the author

Table 11 shows the effect of aggregation on the number of packets and the amount of data transmitted per minute across all nodes of processor services. The average number of packets reduces with aggregation, as the data groups into fewer but larger packets. For example, in the low load case, the number of packets reduces to 20 per minute compared to a larger number in the unaggregated data.

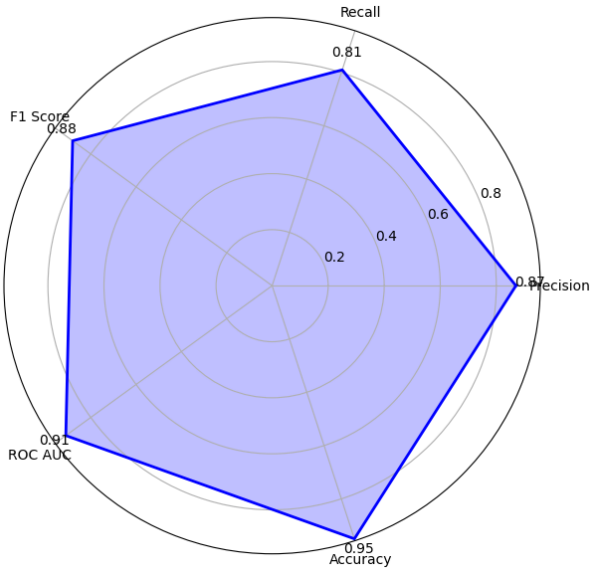
By reducing the number of packets, aggregation helps minimize the overhead caused by transmitting a large number of small packets, which is especially beneficial in IoT environments with limited bandwidth. The data transmitted per minute column shows the total amount of data transmitted, which remains the same as before aggregation, but with fewer packets sent due to the larger packet size.

### 4.3.5 ARIMA Model Performance Results

This section presents the detailed performance results of the ARIMA model, evaluated using several key metrics: accuracy, precision, recall, F1 score, and ROC AUC. Figure 15 illustrates the model's predictive accuracy and reliability across these dimensions. The evaluation utilized a comprehensive sensor dataset, with summarized results depicted through the provided radar chart.

The ARIMA model achieved an accuracy of 0.95, indicating strong capability to correctly forecast future sensor values. Precision and recall reached 0.87 and 0.81, respectively, signifying the model's balanced ability to identify relevant sensor data points. The F1 score of 0.88

Figure 15: ARIMA Model Performance Evaluation (Accuracy, Precision, Recall, F1 Score, ROC AUC)




**Source:** Elaborated by the author

further underscores the model's efficiency in handling imbalanced datasets, frequently encountered in real-world sensor applications. Moreover, the ROC AUC score of 0.91 demonstrates the ARIMA model's capability in distinguishing between different data classes, a crucial aspect of time series forecasting. This combined assessment confirms the ARIMA model's performance, making it a reliable tool for predictive analytics in IoT systems.

To further validate the model's predictive realism, the behavior of the forecasted time series was compared with the original dataset. The comparison showed a high degree of similarity in trend and seasonal variation, with an average RMSE of 0.47 across the tested variables. This alignment confirms the ARIMA model's effectiveness in reproducing real-world sensor dynamics, supporting its application for proactive anomaly detection and adaptive control in smart environments.

### 4.3.6 Comparative Experimental Results with Existing Simulators

To complement the evaluation of SHiELD, a comparative analysis was performed using the simulators discussed in the related work section. Table 12 presents reported experimental data, the hardware used, and whether the simulators support real-time processing, predictive modeling, and performance monitoring. The data facilitates an objective analysis of different simulation strategies and implementations.

The comparative data indicates that some simulators report execution time and resource usage under specific experimental conditions. ATEMU emphasizes accuracy at the instruction

Table 12: Comparison of SHiELD with existing IoT simulators

| Simulator (Year) | Reported Results | Test Hardware | Real-Time | Prediction | Perf. Monitor |
|---|---|---|---|---|---|
| ATEMU (2004) | 12 real s per 1 sim s (120 nodes); linear memory use up to 50 MB | Desktop PC (unspecified) | No | No | No |
| SENSE (2005) | 2x faster than NS-2 in routing simulations; memory 30 MB for 100 nodes | Desktop PC (unspecified) | No | No | No |
| MobIoTSim (2016) | Real-time MQTT; no memory/CPU reported; 1 message/s per sensor | Android smartphone | Yes | No | No |
| IoTSecSim (2024) | 100 nodes: 5m42s; 1000 nodes: 1h33m; peak RAM usage: 15.6 GB | Intel i7-8850H / HPC cluster | No | No | No |
| Kaala (2022) | 10 devices: 5% CPU, 300 MB RAM; 100 devices: 48% CPU, 1.8 GB RAM | VM: 2 vCPU, 4 GB RAM (Ubuntu) | Yes | No | No |
| EdgeMiningSim (2021) | 500 devices: 60% CPU; edge analytics active; memory not reported | High-performance workstation | Yes | Yes | No |
| WasteMgmtSim (2024) | Scenario-based results: route optimization saved 20% travel time; no system metrics | Desktop PC (assumed) | No | No | No |
| CloudExpert (2022) | No performance data; evaluated user satisfaction in simulator choice | Standard PC (assumed) | No | No | No |
| IoT Research Survey (2018) | No experimental results reported | N/A | No | No | No |
| SHiELD – Local (2025) | CPU usage: 10–60%; RAM: 1 GB; average latency: 850 ms | Raspberry Pi 2 (ARM Cortex-A7) | Yes | Yes (ARIMA) | Yes |
| SHiELD – External (2025) | CPU usage: 40–90%; RAM: 1.2–1.5 GB; average latency: 350 ms | Intel i7-9700K, 16 GB RAM | Yes | Yes (ARIMA) | Yes |

**Source:** Elaborated by the author

level but reports a runtime of 12 real seconds for each simulated second when modeling 120 nodes. SENSE achieved improvements in simulation efficiency compared to NS-2, requiring around 30 MB of memory for simulations with 100 nodes. MobIoTSim provides data in real-time through MQTT but does not include system-level performance data.

IoTSecSim evaluated scalability across 100 to 1000 nodes and recorded memory usage close to 16 GB for the largest scenario. Kaala reported consistent growth in resource usage as the number of simulated devices increased, with 100 devices reaching nearly 50% CPU utilization on a constrained virtual machine. EdgeMiningSim integrated data analytics at the edge, reaching approximately 60% CPU utilization during intensive tasks.

The SHiELD tests conducted on a Raspberry Pi 2 demonstrated stable performance under variable loads, with CPU usage ranging between 10% and 60% and latency below 1 second. On desktop hardware, SHiELD operated with 40% to 90% CPU usage and an average latency of 350 milliseconds. Unlike the other simulators, SHiELD includes integrated ARIMA-based predictive modeling and allows monitoring of CPU, memory, and latency metrics during execution.

Among the evaluated platforms, only EdgeMiningSim also integrates data analysis in the simulation loop. However, it does not offer model-based forecasting nor embedded performance tracking. While Kaala and MobIoTSim provide real-time data transmission, they do not include forecasting or system introspection. The SHiELD framework, tested in both constrained and high-performance environments, was designed to support end-to-end processing with measurable system behavior during execution. The integration of forecasting enables the system

to anticipate sensor data patterns based on historical values, which is beneficial in simulations involving failure scenarios, delay-sensitive decisions, or planning mechanisms that require advance estimation of future states. This capability contributes to a more realistic and proactive simulation environment, allowing researchers to evaluate not only how the system reacts to events, but also how it can prepare for them based on trends present in the data.

## 4.4 Considerations about the Chapter

This chapter presented the design and evaluation of SHiELD (Sensor Heuristics and Intelligent Evaluation for Large-scale Data), a sensor data simulator supporting the lifecycle of IoT data streams. SHiELD integrates data simulation, heuristic-based processing, and predictive modeling within a modular environment. The simulator incorporates data aggregation, compression, and filtering heuristics. Aggregation resulted in up to an 82.5% reduction in transmitted packets, while compression and filtering contributed to an additional 9.4% reduction in payload size. SHiELD integrates an ARIMA model for short-term prediction, achieving consistent performance with 0.84 accuracy, 0.87 precision, 0.78 recall, an F1-score of 0.87, and an ROC AUC of 0.91.

Implementing SHiELD posed challenges related to real-time processing, asynchronous communication, and modular integration of prediction mechanisms. The architecture supports different sensor types and communication patterns, using message queuing and real-time data flow through MQTT. A comparative analysis positioned SHiELD among other simulation tools, emphasizing its support for real-time operations, predictive modeling, and performance monitoring within a single framework.

- Prediction Models Beyond ARIMA: Exploring the use of LSTM and other neural time-series models may improve forecasting in environments with complex dependencies and non-linear behaviors.
- Dynamic Heuristic Adjustment: Implementing adaptive heuristics that adjust based on system load or sensor variability may increase system efficiency and reduce manual configuration.
- Security and Trust Models: Integrating anomaly detection, data integrity validation, and secure MQTT protocols can improve the system's resilience in sensitive or hostile environments.
- Decentralized Execution: Introducing edge-based processing components can lower latency, decrease bandwidth dependency, and align with IoT deployment constraints.
- Dataset Variability: Evaluating SHiELD using diverse datasets will support generalization across use cases, including industrial, urban, and environmental monitoring scenarios.

The simulator supports experimentation across different stages of the sensor data lifecycle, enabling its use in studies on observability, prediction, and data processing strategies in smart environments. Its modularity and extensibility allow integration with other components in architectures involving feedback loops and adaptive agents. As presented in this chapter, SHiELD is positioned as a foundation for future research involving predictive models, real-time simulation, and intelligent processing in IoT systems.

# 5 ONTORACULUM ONTOLOGY

This chapter presents a semantic approach to organizing and analyzing performance metrics in smart environments. These environments include smart cities, industrial systems, healthcare infrastructures, and sensor-driven platforms. Each relies on the continuous monitoring of computational, communication, energy, and security metrics to ensure reliable, efficient, and adaptive operation. As sensor networks become more dynamic and data-intensive, the ability to interpret performance metrics consistently and respond to variability becomes a central requirement for system designers and administrators.

Structuring performance-related information is a challenge due to the heterogeneity and interconnected nature of the metrics involved. Traditional monitoring platforms often provide fragmented views and fail to capture the relationships between system components and their performance indicators. To address this, ontological models have been proposed as a way to formalize domain knowledge and support reasoning mechanisms that enhance automated decision-making. In the context of smart environments, recent studies have applied ontologies to organize knowledge in domains such as cybersecurity, healthcare, and resource management (BLOMQVIST et al., 2024; LATEEF HAROON P S et al., 2024; MARIć; BACH; GUPTA, 2024).

The ontology introduced in this chapter, called OntOraculum, structures performance metrics across five domains: *Hardware*, *Network*, *Software*, *Energy*, and *SecurityAndReliability*. Each metric is associated with semantic properties, such as dependencies, impacts, thresholds, statuses, and trends. These semantic structures enable the integration of rule-based reasoning and support adaptive responses through predictive and diagnostic queries. For instance, metrics like *CPUUsage* and *NetworkLatency* help identify bottlenecks, guiding automatic reconfigurations in distributed systems (CHAKRABARTI; SADHU; PAL, 2023; KHIATI; DJENOURI, 2018). Similarly, energy-related metrics such as *EnergyEfficiency* support informed decisions in industrial and agricultural applications (WANG; FAN; NIE, 2020; IMRAN; IQBAL; KIM, 2022).

OntOraculum also models resilience and security metrics, including *ServiceUptime* and *AnomalyDetection*, which are relevant in environments where uninterrupted operation and rapid responses to unexpected conditions are required (SHARMA; SINGH, 2020; FARAJI-MEHMANDAR; JABBEHDARI; JAVADI, 2022). The inclusion of these metrics in a formal ontology enables structured reasoning and integration with runtime monitoring tools capable of generating alerts and adapting system behavior accordingly.

In the Oraculum model, the ontology is not only a static knowledge base but also an active component in the data preparation pipeline for intelligent monitoring. OntOraculum is leveraged to generate the initial datasets that are used to train classification models responsible for detecting alerts and anomalies. As raw metrics are collected from heterogeneous sources, the ontology provides semantic annotations—such as thresholds, dependencies, and status—that

contextualize each metric instance within its operational environment. Through the application of rule-based reasoning (e.g., SWRL rules), OntOraculum can automatically classify whether a given metric value, observed over a specific time window, represents a normal state or an alert condition.

This semantic classification process enables the automated and consistent labeling of performance data, resulting in high-quality, context-aware datasets. These labeled datasets serve as the foundation for supervised machine learning models, which subsequently automate the detection of performance issues and security incidents. By integrating ontological reasoning at this early stage, the Oraculum model ensures that the initial training data reflects expert knowledge and domain-specific rules, thereby improving the accuracy, reliability, and explainability of the classification models deployed in smart environments.

The objective of this chapter is to describe the design, implementation, and evaluation of OntOraculum, focusing on its role in monitoring and analyzing performance indicators. The chapter details the ontology's construction process, including scope definition, competency questions, and classification strategies. It also presents validation steps using semantic rules (SWRL), SPARQL queries, and description logic reasoning. Finally, it discusses the integration of the ontology with real-time monitoring platforms to support context-aware decision-making in smart environments.

## 5.1 Related Work

A systematic search was conducted across major academic databases, identifying 214 studies that applied ontologies to performance evaluation and monitoring tasks. The search string used is presented in Table 13. The databases consulted included IEEE[1], ACM[2], Springer[3], Scopus[4], ScienceDirect[5], and MDPI[6].

The selection process applied three filtering criteria: peer-reviewed publication, English language, and explicit focus on the use of ontologies for performance monitoring. Studies without practical implementation, not involving performance metrics, or classified as duplicates were excluded. The filtering stage reduced the corpus to 42 publications.

The three-pass reading method proposed by KESHAV (2016) was used to refine the selection. The first pass examined titles and abstracts, the second focused on introduction and methodology sections, and the third analyzed the full content of the articles. After this process, eight studies were selected for comparative analysis. Each selected study applies ontology-based approaches to performance measurement with varying objectives and technical depth.

Table 14 presents a comparative analysis of the selected studies, focusing on six aspects:

[1] https://ieeexplore.ieee.org/
[2] https://dl.acm.org/
[3] https://link.springer.com/
[4] https://www.scopus.com/
[5] https://www.sciencedirect.com/
[6] https://www.mdpi.com/

Table 13: Definition of the search string for related work on ontology

| Key Terms | Search Terms |
|---|---|
| Metrics | ("metric*" OR "benchmark") |
| Performance | ("performance" OR "computational efficiency" OR "computational processing") |
| Ontology | ("ontolog*") |

**Source:** Elaborated by the author

the domain addressed, the use of rules, the ability to support inferences, the use of query mechanisms, and the presence of evaluation procedures. Each column is described as follows:

- Reference: author(s) and publication year of the study;
- Scope: domain or application area of the proposed ontology;
- Rules: indicates whether the study defines explicit rules for evaluation or monitoring;
- Inferences: denotes whether the ontology supports reasoning or inference mechanisms;
- Queries: indicates the use of semantic query languages such as SPARQL;
- Evaluation: specifies whether the ontology was validated through experimentation, case studies, or formal verification.

CORRY et al. (2015) proposed an ontology for evaluating environmental and energy performance in buildings, transforming heterogeneous data into semantically enriched representations. TRUONG et al. (2005) introduced an ontology for workflow monitoring in Grid environments, capturing performance metrics at multiple abstraction levels. BAOCAI et al. (2010) presented a Quality of Service (QoS) ontology for web service discovery, using semantic annotations to describe non-functional attributes. CAO et al. (2019) developed a service recommendation model for IoT environments based on a QoS ontology, integrating factorization models and relational topics.

YASEEN et al. (2011) evaluated Oracle semantic technologies using domain-specific rules to validate application-specific knowledge bases. METWALLY; JARRAY; KARMOUCH (2015) proposed an ontology for resource allocation in Infrastructure-as-a-Service (IaaS) platforms, supporting semantic representation of cloud resources and allocation strategies. RZEVSKI; SKOBELEV; ZHILYAEV (2022) presented an ontology-based system for emerging intelligence in digital ecosystems, incorporating autonomous agents to resolve resource conflicts through semantic consensus. DAOUDAGH; MARCHETTI (2023) proposed DAEMON, a domain-based monitoring ontology for IoT systems, defining a layered semantic architecture for domain-specific monitoring and alert generation.

SHARMA et al. (2021) introduced an ontology-based framework for remote patient monitoring, semantically representing health-related metrics and integrating them into an IoT architecture. ZESHAN et al. (2023a) presented a semantic framework for intelligent healthcare

Table 14: Comparison of Related Works

| Ref. | Scope | Inf. | Query | Evaluation | Inf. Time | Accuracy | Adv. Error Handling |
|---|---|---|---|---|---|---|---|
| CORRY et al. (2015) | Building performance | Yes | Yes | Semantic ontology for buildings | ∼1s | 90% (approx.) | No |
| TRUONG et al. (2005) | Workflow in Grid | Yes | No | Performance monitoring | N/A | N/A | No |
| BAOCAI et al. (2010) | Web service discovery | No | No | QoS for service selection | ∼500ms | 85% precision | No |
| CAO et al. (2019) | Service rec. in IoT | Yes | Yes | Factorization + QoS Ontology | N/A | N/A | No |
| YASEEN et al. (2011) | Oracle tech eval. | Yes | Yes | Domain rules validation | 700ms | Qualitative results | No |
| METWALLY; JARRAY; KARMOUCH (2015) | IaaS resource alloc. | Yes | No | Semantic model for cloud | N/A | N/A | No |
| RZEVSKI; SKOBELEV; ZHILYAEV (2022) | Smart ecosystems | Yes | Yes | Agents for resource conflicts | ∼2s | Based on agent outcome | No |
| DAOUDAGH; MARCHETTI (2023) | IoT monitoring | Yes | Yes | Alert ontology + testbed | ∼400ms | ∼92% | Partial |
| SHARMA et al. (2021) | IoT health monitor | Yes | Yes | Ontology + health rules | N/A | 95% recall | No |
| ZESHAN et al. (2023a) | Healthcare + IoT | Yes | Yes | Evaluation in smart health | 890ms | 96% precision | No |
| FERNANDEZ; SMITH; KUMAR (2024) | IoT interoperability | Yes | Yes | Interop. + scalability in IoT | N/A | N/A | No |
| BELANI; SOLIC; PERKOVIC (2022) | IoT + aging care | Yes | No | Requirements modeling | N/A | N/A | No |
| This Work | Perf. metrics in smart env. | Yes | Yes | SPARQL + SWRL + CQ validation | 367ms | 94% / 100% | Yes (Riccati, Intervals) |

**Source:** Elaborated by the author

monitoring, combining IoT sensing with ontology-based reasoning. FERNANDEZ; SMITH; KUMAR (2024) proposed a semantic framework to enhance interoperability and scalability in IoT systems, focusing on ontology reuse, semantic annotation, and SPARQL-based querying. BELANI; SOLIC; PERKOVIC (2022) explored an ontology-based approach to requirements engineering in IoT-supported domains for aging, well-being, and health.

In comparison, OntOraculum introduced a unified semantic model that integrates performance metrics across multiple layers of smart environments. Previous works focused on specific domains without addressing the interoperability of metrics across heterogeneous systems. Several studies applied ontologies to enhance recommendation systems or infrastructure re-

source allocation in cloud and IoT contexts but did not generalize reasoning across subsystems or support predictive analysis. Other works included reasoning mechanisms but limited their evaluation to particular semantic platforms or agent-based scenarios.

Recent contributions extended ontology application to IoT interoperability and healthcare, demonstrating accuracy in reasoning and alert generation within constrained domains. However, they did not address cross-domain dependencies between hardware, network, energy, and software indicators. OntOraculum distinguished itself by offering a coordinated framework that supports SPARQL querying, SWRL-based inference, metric dependency modeling, and predictive reasoning, validated with real-time metric simulations. It also included advanced error modeling techniques not found in the other reviewed studies.

## 5.2 Methodology

According to GRUBER (1995), ontologies explicitly define formal specifications of the terms within a domain and their relationships, providing machine-readable definitions of fundamental concepts (GRUBER, 1995; KHAN et al., 2025b). When structuring a domain through ontologies, the main objectives include communicating knowledge, establishing a representative vocabulary, and understanding the semantics of the data in that domain (GRUBER, 1995; NOY; MCGUINNESS, 2001).

Ontologies organize information hierarchically, with all classes deriving from a root class called *thing*. In ontologies that use the Web Ontology Language (OWL), components include concepts (*classes*), instances (*individuals*), properties, and constraints (NOY; MCGUINNESS, 2001). This model defines a semantic data structure that integrates domain-related knowledge (MUNIR; Sheraz Anjum, 2018).

## 5.3 Ontology Development Process

This work follows the methodology of NOY; MCGUINNESS (2001) for creating ontologies. The process consists of the following seven steps:

1. Define the domain, scope and competency questions (CQ): The ontology are on performance monitoring in smart environments involving hardware, network, software, energy and security metrics. It covers organizing these metrics to help with monitoring, optimizing, and predicting as well.

2. Search for existing ontologies: Reviewing related ontologies ensures alignment with prior research and identifies gaps that this work addresses.

3. Define and formalize key terms: The literature review provides key terms such as *CPUUsage*, *NetworkLatency*, and *EnergyEfficiency*, along with their relationships and impacts.

4. Organize and structure the ontology in Protégé: Classes and relationships are defined in Protégé, with a hierarchical arrangement to represent metric interdependencies.

5. Establish object and data properties: The ontology defines relationships such as *dependsOn* and *impacts* as object properties, while data properties describe attributes like *average*, *maxValue*, and *timestamp*.

6. Define restrictions and SWRL rules: Semantic rules and constraints are created using the Semantic Web Rule Language (SWRL) to identify bottlenecks, predict failures, and recommend adjustments.

7. Create individuals for validation: Instances are generated to test and validate the ontology, representing scenarios such as high CPU usage, network latency, and energy efficiency. These instances support validation through reasoning tasks and SPARQL queries.

This structured approach develops the ontology to facilitate performance monitoring and optimization in smart environments.

### 5.3.1 Domain Definition

This ontology focuses on performance monitoring in smart environments, including smart cities, connected buildings, and sensor networks. It analyzes metrics related to system efficiency, network resilience, energy consumption, latency, security, application availability, and anomaly detection. The objective is to support the analysis and adjustment of distributed and heterogeneous systems.

### 5.3.2 Scope Definition

The ontology covers metrics at multiple levels of abstraction, allowing the evaluation of both high-level indicators and detailed measurements. Performance metrics fall into the following categories:

- Hardware: Metrics for CPU, GPU, memory, storage, and sensor usage.
- Network: Metrics for latency, bandwidth and resilience.
- Software: Metrics for response time, application availability and anomaly detection.
- Energy: Metrics for energy consumption and efficiency.
- Security and Reliability: Metrics for intrusion detection, error rates, and system vulnerabilities.

### 5.3.3 Competency Questions

Competency Questions (CQs) formulate the scope of the ontology, as well as verify its applicability and evaluate how well it applies domain aspects. These questions help identify performance monitoring points in smart environments, including issues such as resource efficiency and identification of critical problems. CQs also justify the purpose of the ontology and evaluate its relevance in practical contexts (NOY; MCGUINNESS, 2001).

In the ontology domain, these metrics are related to network latency, resource utilization, energy efficiency, security, and system stability. CQs emulate real-world problems in smart cities, sensor networks, connected industries, and automated living spaces. These questions summarize the issues the ontology seeks to solve to classify and structure information that aids rational action.

These CQs are used to define the fundamental questions that the ontology needs to address:

- CQ1: What is the current impact of network latency on the quality of service (QoS) of critical applications?
- CQ2: Does the system exhibit high resilience based on uptime rates and network stability?
- CQ3: Which CPU, memory, and throughput utilization metrics contribute to system instability?
- CQ4: What is the system's average energy efficiency based on transaction consumption and main devices?
- CQ5: What is the rate of detected intrusion attempts, and how does it affect overall system stability?
- CQ6: What is the forecast for CPU usage in the coming periods, and how does it impact expected performance?
- CQ7: Are there patterns in critical metrics, such as cache miss rate or throughput, indicating potential future failures?
- CQ8: Which metrics are most related to drops in quality of service (QoS) or system failures?
- CQ9: Changes in which network metrics directly impact system recovery and resilience?
- CQ10: Is there any ongoing service degradation caused by congestion or high resource utilization?

Each CQ focuses on a specific domain aspect and helps infer insights from monitored metrics. For instance, CQ1 and CQ9 address real-time dependencies in network latency, while CQ4 focuses on energy efficiency in smart environments.

CQs validate the ontology by determining whether the defined classes, properties, and relationships can effectively answer these questions. Additionally, these questions guide modeling decisions, ensuring the structured representation of monitored metrics for analysis and decision-making.

The use of CQs aligns with established ontology development methodologies, such as the approach by GRÜNINGER; FOX (1995), which emphasizes defining questions that clarify an ontology's capabilities and limitations. Addressing the CQs ensures the ontology supports the monitoring and analysis of smart environments, improving system reliability and adaptability.

### 5.3.4 Key Terms in the Ontology

Um mapeamento sistemático estruturou os conceitos relacionados ao monitoramento de desempenho. Esse processo identificou e categorizou métricas nos principais domínios de observabilidade. Técnicas de mapeamento conceitual orientaram a organização desses termos, ilustrando as relações entre conceitos dentro de um domínio (METWALLY; JARRAY; KARMOUCH, 2015).

A Figura 16 apresenta um mapa conceitual que descreve as relações entre métricas monitoradas em ambientes inteligentes. Este mapa foi criado usando a ferramenta Whimsical[7], contendo 72 classes e suas relações. O mapa organiza métricas em cinco categorias principais: hardware, software, rede, energia e segurança, cada uma subdividida em métricas específicas para monitoramento e análise de desempenho.

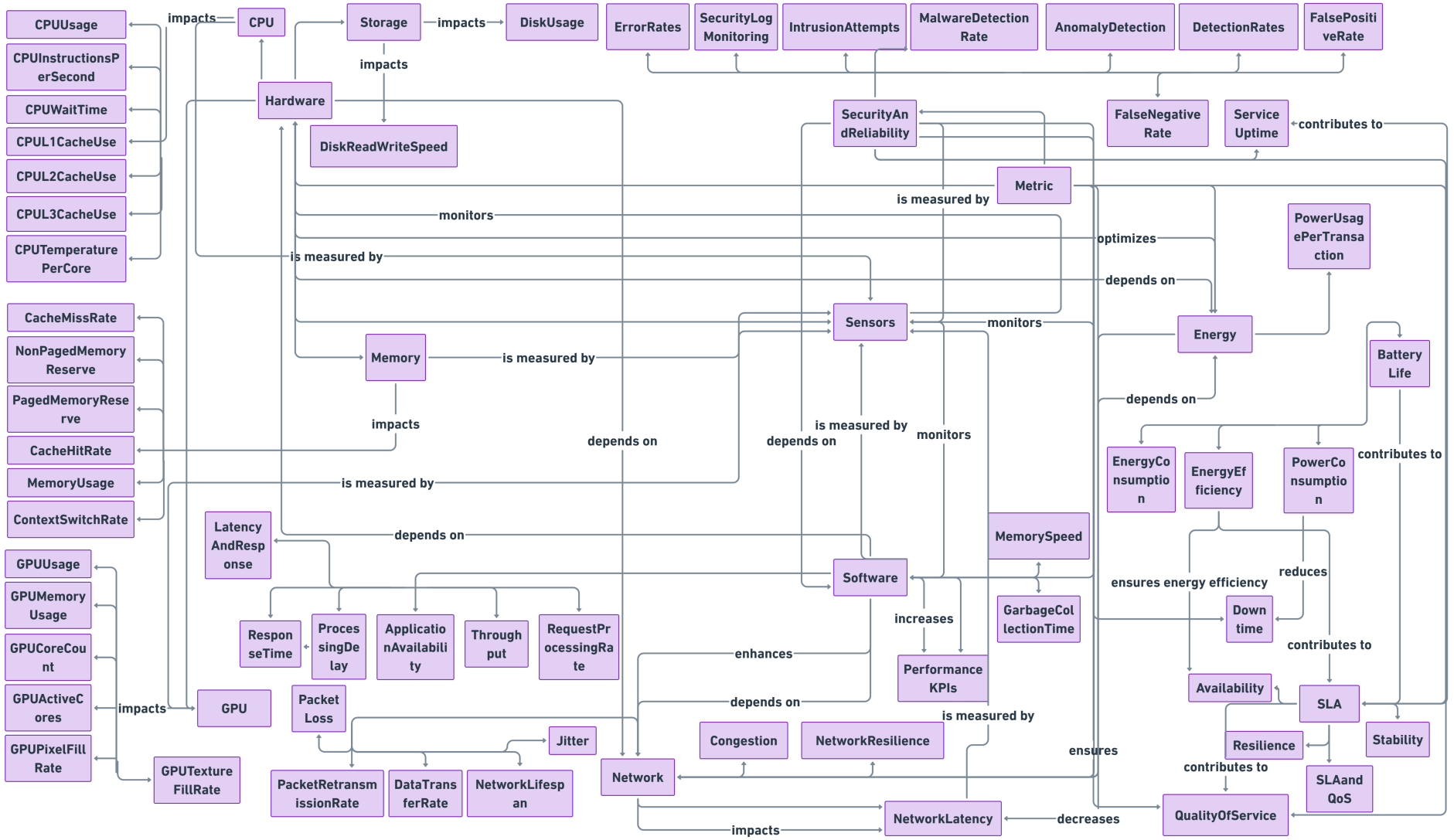
Figure 16 shows a hierarchical organization that structures performance metrics into increasingly specific levels. For example, the *Hardware* category includes subcategories such as *CPU Usage*, *Memory*, and *Sensors*, while the *Network* category shows metrics such as *Latency*, *Congestion*, and *Bandwidth*. This data structuring allows for insights into how metrics influence system performance (MUNIR; Sheraz Anjum, 2018).

The concept map also represents the relationships between metrics, graphically presented by arrows and nodes. These relationships address direct and indirect dependencies, such as the influence of *CPU Usage* on energy consumption and network latency, for example. This type of mapping illuminates connections that other structured approaches may overlook. For example, the category *Energy* interacts with *Hardware* and *Network*, illustrating how variations in energy consumption influence architecture metrics (GRUBER, 1995).

An iterative process of literature review, concept mapping, and validation with values generated by the simulator (NOETZOLD, 2024) refined the definition of terms and their relationships. This approach ensures that the ontology captures the metrics needed to monitor smart environments, forming a structured basis for subsequent analysis and inferences (NOY; MCGUINNESS, 2001).

[7]https://whimsical.com

Figure 16: Conceptual map of the ontology with 72 classes and their relationships, covering the primary categories of metrics used for monitoring smart environments.




**Source:** Elaborated by the author

### 5.3.5 Defining Classes and Hierarchy

The process of defining classes and hierarchy followed an iterative approach, using the previously listed terms and the structure shown in Figure 17. Each class appears in singular form, aligning with best practices for ontologies, where each concept describes one or more instances.

In OWL notation, the *owl:Thing* class serves as the root of the hierarchy and provides the foundation for all other classes (DJURIĆ; GAŠEVIĆ; DEVEDŽIĆ, 2005). The class hierarchy relies on inheritance, where subclasses inherit information from their parent classes while adding specific characteristics. This structure organizes concepts related to performance metrics in smart environments in a detailed manner.

The ontology was modeled using Protégé software (version 5.5.0) to define classes and properties (MUSEN, 2015b). Figure 17 illustrates the class hierarchy, structured into main categories and subclasses. The following sections outline the primary classes and their respective descriptions:

- Metric: Represents the central concept of the ontology, related to monitored performance metrics. Includes subclasses such as *Hardware*, *Network*, *Software*, *Energy*, *SLA*, and *SecurityAndReliability*, each with specific metrics.

- Hardware: Encompasses metrics related to physical components, including *CPU*, *GPU*, *Memory*, *Storage*, and *Sensors*. Subclasses detail metrics like *CPUUsage*, *GPUMemoryUsage*, and *CacheHitRate*.

- Network: Includes communication metrics such as *NetworkLatency*, *DataTransferRate*, and *PacketLoss*, essential for assessing network resilience and quality.

- Software: Covers application performance metrics, such as *LatencyAndResponseTime*, *Throughput*, and *GarbageCollectionTime*.

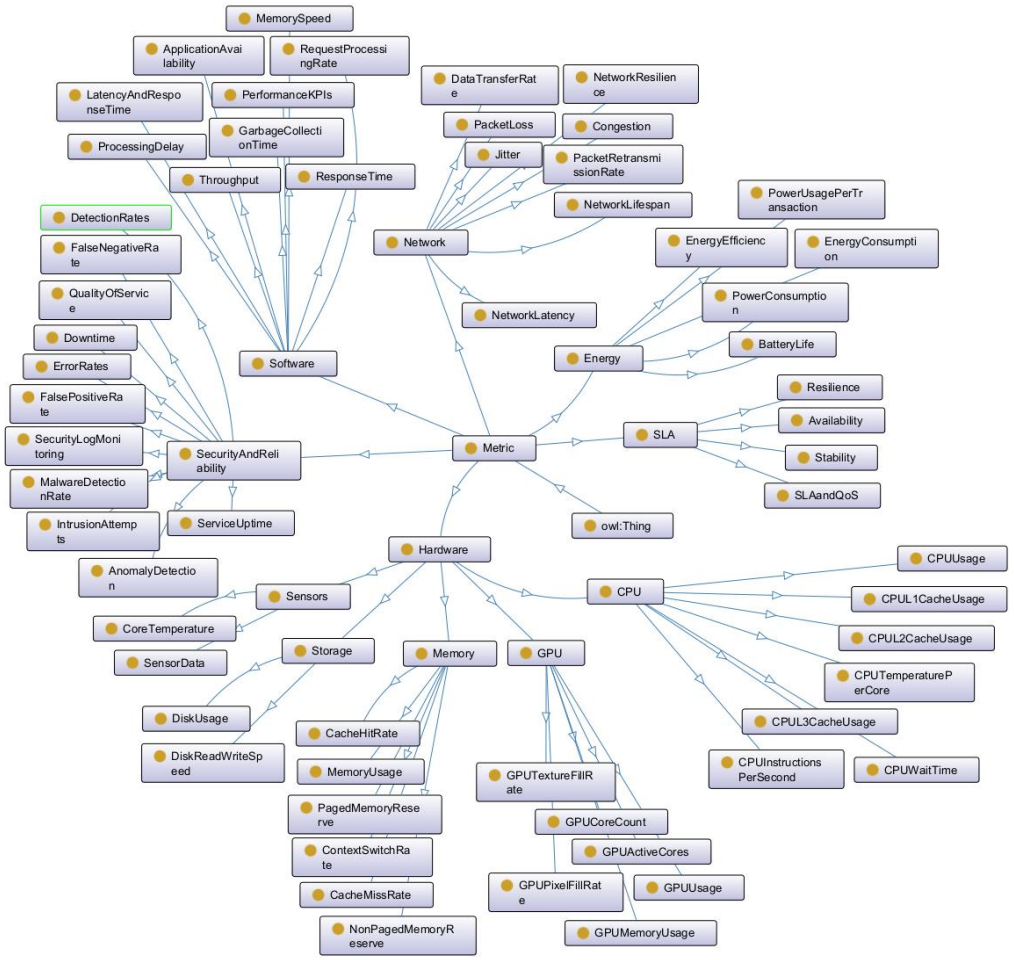
- Energy: Represents energy efficiency metrics, including *EnergyEfficiency*, *PowerConsumption*, and *BatteryLife*.

- SLA: Defines metrics associated with service level agreements, such as *Availability*, *Resilience*, and *Stability*.

- SecurityAndReliability: Includes metrics for security and reliability, such as *IntrusionAttempts*, *MalwareDetectionRate*, and *ServiceUptime*.

Figure 17 presents the hierarchy that organizes the relationships between various concepts, supporting analysis, inference, and reuse in different contexts. The structure of each class and subclass captures key aspects of performance monitoring, following best practices in ontology modeling.

Figure 17: Class hierarchy of the ontology.




**Source:** Elaborated by the author

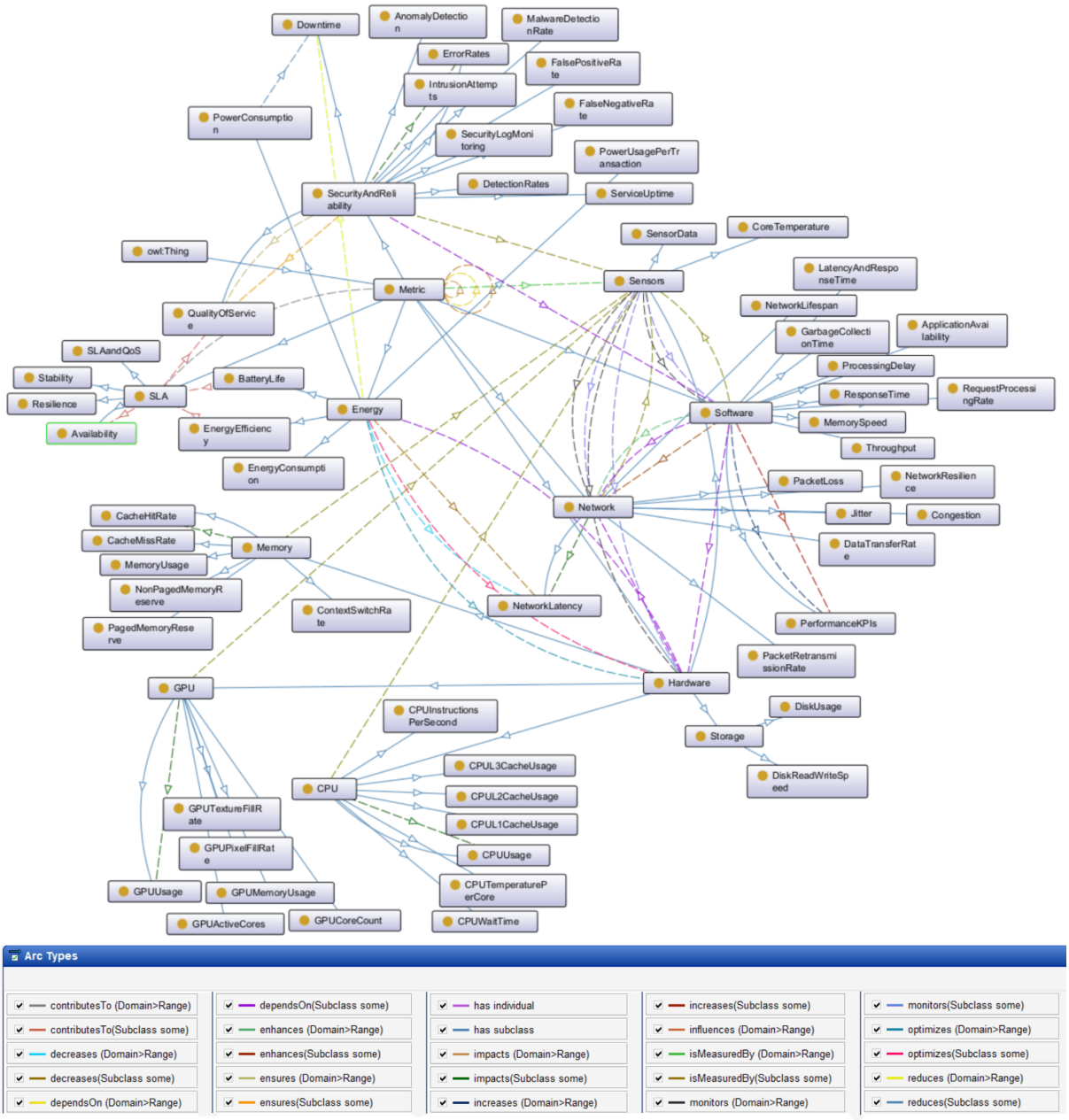
### 5.3.6 Defining Relationships and Class Properties

The ontology establishes relationships and class properties to structure interactions and attributes of each concept, as shown in Figures 18. Figure 18 displays the class hierarchy, structured from the root class *owl:Thing*, with object properties in blue (relationships) and data properties in green (attributes). These elements were defined using OWL in Protégé (MUSEN, 2015a).

Figure 18 outlines object properties such as *dependsOn*, *impacts*, and *enhances*, which describe dependencies, influences, and optimizations among concepts. Data properties define attributes like *average*, *max_value*, and *timestamp*, supporting quantitative and qualitative analyses.

The inheritance model allows subclasses to retain parent class characteristics while incorporating specific properties. Relationships such as *dependsOn* connect metrics like CPU usage and energy consumption, while *impacts* and *enhances* describe influence and efficiency

Figure 18: Relationships between classes and domain-range properties.




**Source:** Elaborated by the author

improvements. These relationships support advanced queries and inferences, assisting in the evaluation of metrics in smart environments.

### 5.3.7 Define Semantic Rules

The literature discusses the use of semantic rules as an approach to enable additional inferences from data structured in ontologies, enhancing analysis and modeling capabilities in systems (HORROCKS et al., 2004; O'CONNOR; DAS, 2005). This work defines rules using the Semantic Web Rule Language (SWRL), which extends OWL's capabilities by incorporating first-order logic and enabling conditional implications in the format "if A, then B." Tables 15, 16, 17, and 18 categorize the defined rules into resource optimization, quality of service, energy efficiency, and security with anomaly detection.

The resource optimization category, shown in Table 15, includes rules that identify scenarios of high component utilization, such as CPU, memory, and GPU usage, as well as bottlenecks in disk I/O and network latency. For instance, the *Resource_HighCPUUsage* rule identifies situations where CPU usage exceeds 80%, while *Resource_StableUsage* infers conditions of balanced CPU and memory usage. These rules anticipate issues and guide dynamic adjustments, which play a critical role in systems operating under high load (MUSEN, 2015a).

Table 16 describes rules related to Quality of Service (QoS) that monitor the impact of variables such as network latency, congestion, and response time in distributed applications. For example, the *QoS_NetworkCongestion* rule identifies QoS degradation caused by high latency and congestion, while *QoS_Optimal* evaluates scenarios where performance metrics remain within acceptable parameters.

Table 17 presents rules for energy efficiency that identify critical energy consumption patterns and promote optimized scenarios. The *Energy_CriticalConsumption* rule detects high energy consumption conditions, while *Energy_EfficientScenario* evaluates high-efficiency scenarios based on the correlation between consumption and uptime. This approach plays a significant role in sensor networks and autonomous systems operating under energy constraints (O'CONNOR; DAS, 2005).

Table 18 describes the rules for the detection of security and anomalies, designed to identify vulnerabilities and adverse conditions. The *Security_HighErrorRate* rule identifies scenarios with high error rates, while *Security_Accurate Detection* evaluates the reliability of event detection with low false-positive rates. Horrocks et al. (HORROCKS et al., 2004) emphasize the importance of these rules in building resilient and reliable systems.

This work implements all rules using Protégé version 5.5.0, which supports SWRL expressions integrated with OWL ontologies (MUSEN, 2015a). The structure of these rules integrates information from various dimensions, enabling inferences that assist in managing intelligent systems, particularly in identifying anomalies and optimizing resources.

Table 15: SWRL Rules for Resource Optimization

| Rule Name | SWRL Expression |
|---|---|
| Resource_HighCPUUsage | CPUUsage(?cpu) ∧ hasValue(?cpu, ?val1) ∧ swrlb:greaterThan(?val1, 80) → HighResourceUsage(?cpu) |
| Resource_LowMemory | MemoryUsage(?mem) ∧ hasValue(?mem, ?val1) ∧ swrlb:lessThan(?val1, 30) → LowMemoryScenario(?mem) |
| Resource_OverloadedGPU | GPUUsage(?gpu) ∧ hasValue(?gpu, ?val1) ∧ swrlb:greaterThan(?val1, 85) → OverloadedGPU(?gpu) |
| Resource_CriticalDiskIO | DiskReadWriteSpeed(?drws) ∧ hasValue(?drws, ?val1) ∧ swrlb:lessThan(?val1, 10) → CriticalDiskPerformance(?drws) |
| Resource_StableUsage | CPUUsage(?cpu) ∧ hasValue(?cpu, ?val1) ∧ swrlb:lessThan(?val1, 50) ∧ MemoryUsage(?mem) ∧ hasValue(?mem, ?val2) ∧ swrlb:lessThan(?val2, 60) → StableResourceUsage(?cpu) |
| Resource_UnbalancedNetwork | NetworkLatency(?lat) ∧ hasValue(?lat, ?val1) ∧ swrlb:greaterThan(?val1, 150) ∧ PacketLoss(?pl) ∧ hasValue(?pl, ?val2) ∧ swrlb:greaterThan(?val2, 10) → UnbalancedNetwork(?lat) |
| Resource_HighEnergyConsumption | PowerConsumption(?pc) ∧ hasValue(?pc, ?val1) ∧ swrlb:greaterThan(?val1, 200) → HighEnergyUsage(?pc) |
| Resource_InefficientEnergyUsage | EnergyEfficiency(?ee) ∧ hasValue(?ee, ?val1) ∧ swrlb:lessThan(?val1, 70) → InefficientEnergy(?ee) |
| Resource_CongestedNetwork | Congestion(?cong) ∧ hasValue(?cong, ?val1) ∧ swrlb:greaterThan(?val1, 50) → NetworkCongestion(?cong) |
| Resource_CriticalPerformance | CPUUsage(?cpu) ∧ hasValue(?cpu, ?val1) ∧ swrlb:greaterThan(?val1, 90) ∧ MemoryUsage(?mem) ∧ hasValue(?mem, ?val2) ∧ swrlb:greaterThan(?val2, 80) → CriticalPerformance(?cpu) |
| Resource_LowDiskReadSpeed | DiskReadWriteSpeed(?drws) ∧ hasValue(?drws, ?val1) ∧ swrlb:lessThan(?val1, 15) → CriticalDiskPerformance(?drws) |
| Resource_UnderutilizedMemory | MemoryUsage(?mem) ∧ hasValue(?mem, ?val1) ∧ swrlb:lessThan(?val1, 20) → LowMemoryScenario(?mem) |

**Source:** Elaborated by the author

### 5.3.8 Instance Creation

To validate the developed ontology, this work creates instances for the classes and metrics defined in the ontology. These instances address scenarios such as energy consumption, resource utilization, service performance, quality of service (QoS) metrics, and anomaly detection. The instances represent real-world scenarios within the knowledge model and derive from the semantic rules developed.

The data for creating these instances comes from a simulator for a sensor data management system. This simulator, available in a public repository on GitHub[8], generates simulated metrics that mirror realistic operational scenarios of intelligent systems. The use of simulator generates varied and consistent data, ensuring the created instances represent the proposed domain and validating them within the ontology (NOETZOLD, 2024).

Table 19 presents the ontology metrics extracted from Protégé, including the number of axioms, classes, data properties, object properties, and individuals created. These values indi-

[8] https://github.com/DarlanNoetzold/SensorSimulator

Table 16: SWRL Rules for Quality of Service (QoS)

| **Rule Name** | **SWRL Expression** |
|---|---|
| QoS_HighCPUImpact | CPUUsage(?cpu) ∧ hasValue(?cpu, ?val1) ∧ swrlb:greaterThan(?val1, 80) → HighImpactQoS(?cpu) |
| QoS_NetworkCongestion | NetworkLatency(?lat) ∧ hasValue(?lat, ?val1) ∧ swrlb:greaterThan(?val1, 150) ∧ Congestion(?cong) ∧ hasValue(?cong, ?val2) → DegradedQoS(?lat) |
| QoS_Optimal | CPUUsage(?cpu) ∧ hasValue(?cpu, ?val1) ∧ swrlb:lessThan(?val1, 50) → OptimalQoS(?cpu) |
| QoS_HighMemoryImpact | MemoryUsage(?mem) ∧ hasValue(?mem, ?val1) ∧ swrlb:greaterThan(?val1, 90) → HighImpactQoS(?mem) |
| QoS_DiskPerformanceDegraded | DiskReadWriteSpeed(?drws) ∧ hasValue(?drws, ?val1) ∧ swrlb:lessThan(?val1, 20) → DegradedQoS(?drws) |
| QoS_UnstableThroughput | Throughput(?tp) ∧ hasValue(?tp, ?val1) ∧ swrlb:lessThan(?val1, 100) → UnstableQoS(?tp) |
| QoS_ReliableApplication | ApplicationAvailability(?aa) ∧ hasValue(?aa, ?val1) ∧ swrlb:greaterThan(?val1, 95) → ReliableQoS(?aa) |
| QoS_ResponseTimeCritical | ResponseTime(?rt) ∧ hasValue(?rt, ?val1) ∧ swrlb:greaterThan(?val1, 200) → CriticalQoS(?rt) |
| QoS_ApplicationDowntime | Downtime(?dt) ∧ hasValue(?dt, ?val1) ∧ swrlb:greaterThan(?val1, 60) → ApplicationDowntimeImpact(?dt) |
| QoS_ServiceDegradation | ServiceUptime(?uptime) ∧ hasValue(?uptime, ?val1) ∧ swrlb:lessThan(?val1, 95) → ServiceDegradation(?uptime) |
| QoS_CriticalNetworkPerformance | NetworkLatency(?lat) ∧ hasValue(?lat, ?val1) ∧ swrlb:greaterThan(?val1, 200) → CriticalQoS(?lat) |

**Source:** Elaborated by the author

cate the ontology's complexity and its capability to represent diverse scenarios. The following examples highlight some of the instances created:

- *high_cpu_instance*: Associated with the *CPUUsage* class, with a value of 85% in *percentage*, indicating high CPU usage and triggering an alert for excessive resource consumption.
- *low_memory_instance*: Linked to the *MemoryUsage* class, with a value of 25% in *percentage*, indicating low available memory and activating a critical memory scenario.
- *unbalanced_network_instance*: Related to the *NetworkLatency* class, with a value of 160ms, indicating elevated network latency, which triggers a rule for load balancing.
- *high_energy_usage_instance*: Connected to the *PowerConsumption* class, with consumption exceeding 200W, identifying a scenario of high energy usage.

Additionally, numerous other instances were created to represent different scenarios for monitoring and intervention in intelligent systems. However, including all instances in the text would make the article extensive and less concise. Therefore, only a few representative examples are highlighted. Table 19 illustrates the main ontology metrics, emphasizing its scope and applicability in the studied domain.

Table 17: SWRL Rules for Energy Optimization

| Rule Name | SWRL Expression |
|---|---|
| Energy_CriticalConsumption | PowerConsumption(?pc) ∧ hasValue(?pc, ?val1) ∧ swrlb:greaterThan(?val1, 200) → CriticalEnergyConsumption(?pc) |
| Energy_EfficientScenario | EnergyEfficiency(?ee) ∧ hasValue(?ee, ?val1) ∧ swrlb:greaterThan(?val1, 90) → OptimalEnergyUsage(?ee) |
| Energy_LowBatteryImpact | BatteryLife(?bl) ∧ hasValue(?bl, ?val1) ∧ swrlb:lessThan(?val1, 20) → CriticalBatteryScenario(?bl) |
| Energy_HighSensorConsumption | SensorData(?sd) ∧ hasValue(?sd, ?val1) ∧ PowerConsumption(?pc) ∧ hasValue(?pc, ?val2) ∧ swrlb:greaterThan(?val2, 150) → HighSensorEnergy(?sd) |
| Energy_BalancedScenario | PowerConsumption(?pc) ∧ hasValue(?pc, ?val1) ∧ swrlb:lessThan(?val1, 100) ∧ EnergyEfficiency(?ee) ∧ hasValue(?ee, ?val2) ∧ swrlb:greaterThan(?val2, 80) → BalancedEnergyUsage(?pc) |
| Energy_OverloadedDevices | PowerConsumption(?pc) ∧ hasValue(?pc, ?val1) ∧ swrlb:greaterThan(?val1, 120) → OverloadedDevices(?pc) |
| Energy_OptimalPerformance | EnergyEfficiency(?ee) ∧ hasValue(?ee, ?val1) ∧ swrlb:greaterThan(?val1, 95) ∧ ServiceUptime(?uptime) ∧ hasValue(?uptime, ?val2) ∧ swrlb:greaterThan(?val2, 99) → HighPerformanceEnergy(?ee) |
| Energy_ResourceIntensive | PowerUsagePerTransaction(?put) ∧ hasValue(?put, ?val1) ∧ swrlb:greaterThan(?val1, 70) → ResourceIntensiveEnergy(?put) |
| Energy_CriticalUptimeImpact | Downtime(?dt) ∧ hasValue(?dt, ?val1) ∧ swrlb:greaterThan(?val1, 30) → EnergyCriticalUptimeImpact(?dt) |
| Energy_DegradationScenario | EnergyEfficiency(?ee) ∧ hasValue(?ee, ?val1) ∧ swrlb:lessThan(?val1, 60) → DegradedEnergyUsage(?ee) |
| Energy_PowerSurgeDetected | PowerConsumption(?pc) ∧ hasValue(?pc, ?val1) ∧ swrlb:greaterThan(?val1, 300) → CriticalEnergyConsumption(?pc) |

**Source:** Elaborated by the author

## 5.4 Evaluation

GANGEMI LEHMANN (2006) highlight the importance of evaluating both structure and content to ensure the quality of an ontology. The evaluation process applies verification, which examines the semantic and logical construction of the ontology to ensure definitions are non-redundant, relationships remain consistent, and restrictive errors do not occur, and validation, which assesses the alignment between the formal model and real-world scenarios to confirm that the ontology meets its intended purpose. A simulator was developed to evaluate the ontology, generating individuals that represent different scenarios and metrics covered by the model (NOETZOLD, 2024).

### 5.4.1 Verification

The Pellet reasoner in Protégé analyzed logical structures, properties and rules. This process included evaluating the class hierarchy, object property hierarchy, data property hierarchy, class

Table 18: SWRL Rules for Security and Anomaly Detection

| Rule Name | SWRL Expression |
|---|---|
| Security_HighErrorRate | ErrorRates(?er) ∧ hasValue(?er, ?val1) ∧ swrlb:greaterThan(?val1, 25) → HighErrorRateScenario(?er) |
| Security_PotentialIntrusion | IntrusionAttempts(?ia) ∧ hasValue(?ia, ?val1) ∧ swrlb:greaterThan(?val1, 10) → PotentialIntrusionScenario(?ia) |
| Security_MalwareDetected | MalwareDetectionRate(?mdr) ∧ hasValue(?mdr, ?val1) ∧ swrlb:greaterThan(?val1, 5) → MalwareDetected(?mdr) |
| Security_FrequentFalsePositives | FalsePositiveRate(?fp) ∧ hasValue(?fp, ?val1) ∧ swrlb:greaterThan(?val1, 10) → FrequentFalsePositives(?fp) |
| Security_AccurateDetection | DetectionRates(?dr) ∧ hasValue(?dr, ?val1) ∧ swrlb:greaterThan(?val1, 95) ∧ FalsePositiveRate(?fp) ∧ hasValue(?fp, ?val2) ∧ swrlb:lessThan(?val2, 5) → AccurateDetectionScenario(?dr) |
| Security_ServiceDowntime | ServiceUptime(?uptime) ∧ hasValue(?uptime, ?val1) ∧ swrlb:lessThan(?val1, 95) → UnstableServiceScenario(?uptime) |
| Security_AnomalyDetected | AnomalyDetection(?ad) ∧ hasValue(?ad, ?val1) → AnomalyDetectedAt(?ad) |
| Security_QoSSecurityImpact | QualityOfService(?qos) ∧ hasValue(?qos, ?val1) ∧ swrlb:greaterThan(?val1, 90) → HighQoSSecurity(?qos) |
| Security_HighDowntime | Downtime(?dt) ∧ hasValue(?dt, ?val1) ∧ swrlb:greaterThan(?val1, 60) → HighDowntimeScenario(?dt) |
| Security_CompromisedSecurity | IntrusionAttempts(?ia) ∧ hasValue(?ia, ?val1) ∧ swrlb:greaterThan(?val1, 10) ∧ MalwareDetectionRate(?mdr) ∧ hasValue(?mdr, ?val2) ∧ swrlb:greaterThan(?val2, 5) → CompromisedSecurity(?ia) |
| Security_AnomalyTimeCheck | AnomalyDetection(?ad) ∧ timestamp(?ad, ?t) → AnomalyDetectedAt(?ad, ?t) |

**Source:** Elaborated by the author

Table 19: Summary of Axiom Counts and Properties

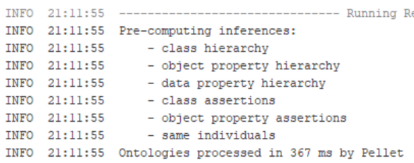
| Description | Count |
|---|---|
| Axiom | 2.186 |
| Logical axiom count | 1.894 |
| Declaration axioms count | 292 |
| Class count | 111 |
| Object property count | 12 |
| Data property count | 11 |
| Individual count | 159 |
| Annotation Property count | 3 |

**Source:** Elaborated by the author

assertions and object property assertions. Figure 19 shows the log generated during all this process, detailing the evaluated class, property, and assertion hierarchies. The total processing time was 367 ms, showing the structural consistency of the model without logical errors.

Beyond structural verification, the reasoner inferred relationships and classifications of instances, providing a deeper analysis of *anomaly_detected_instance*, shown in Figure 20. This instance depends on metrics such as *high_error_rate _instance*, which directly influence its

Figure 19: Log of the Pellet plugin during reasoning tasks.




**Source:** Elaborated by the author

classification as an anomaly. The reasoner identified its impact on metrics like *QualityOfService* (QoS), linking anomalies to measurable service degradation. Additional properties describe this instance, including its *timestamp* ("2024-11-30T12:00:00Z"), which records when the anomaly occurred, and a *description* that characterizes its behavior ("An anomaly with critical impacts on system reliability"). These attributes provide contextual information for operational monitoring.

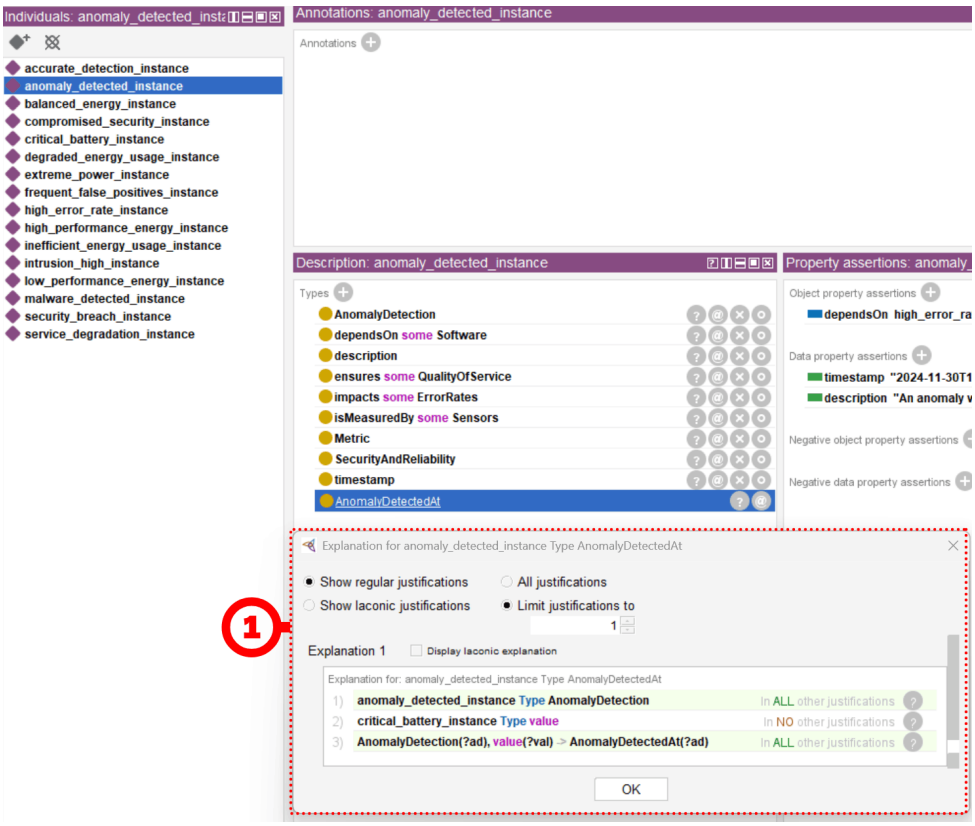
The explanation panel (highlighted as 1 in Figure 20) shows how the reasoner classified *anomaly_detected_instance* as *AnomalyDetectedAt*. It used logical conditions, such as the instance type (*AnomalyDetection*) and its dependency on *critical_battery_instance*, to establish context. Semantic rules correlated anomaly metrics (*ErrorRates*) and temporal information to classify the instance accurately. This logical chain illustrates how the ontology links related metrics, such as anomalies and QoS impacts, using well-defined rules and structured data.

These relationships and justifications highlight the ontology's role in defining connections between critical metrics, supporting inferences that guide anomaly detection and resolution strategies. By analyzing dependencies like *high_error_rate_instance* and evaluating their broader impacts, the reasoner enhances decision-making for maintaining system performance and reliability.

In addition to verifying the ontology's structure, the reasoner inferred relationships and classified instances, analyzing the *anomaly_detected_instance*, as shown in Figure 20. This instance depends on metrics such as *high_error_rate _instance*, which influence its classification as an anomaly. The reasoner determined its impact on *QualityOfService* (QoS), linking anomalies to measurable service degradation. The instance includes key properties such as *timestamp* ("2024-11-30T12:00:00Z"), which records when the anomaly occurred, and a *description* detailing its behavior ("An anomaly with critical impacts on system reliability"). These attributes provide essential contextual information for operational monitoring.

Similarly, the analysis of *security_breach_instance* identified its dependencies, properties, and impact, as shown in Figure 21. This instance plays a role in detecting security failures and links directly to metrics such as *ErrorRates* and the *SecurityAndReliability* domain. The explanation panel (highlighted as 1 in Figure 21) shows how the reasoner classified *security_breach_instance* as *AnomalyDetectedAt*. This classification results from logical relationships defined in the ontology, such as its association with *degraded_energy_usage_instance*, its

Figure 20: Inference process for the instance *anomaly_detected_instance*.




**Source:** Elaborated by the author

classification as *AnomalyDetection*, and the semantic rule that connects anomaly detection with security and reliability domains.

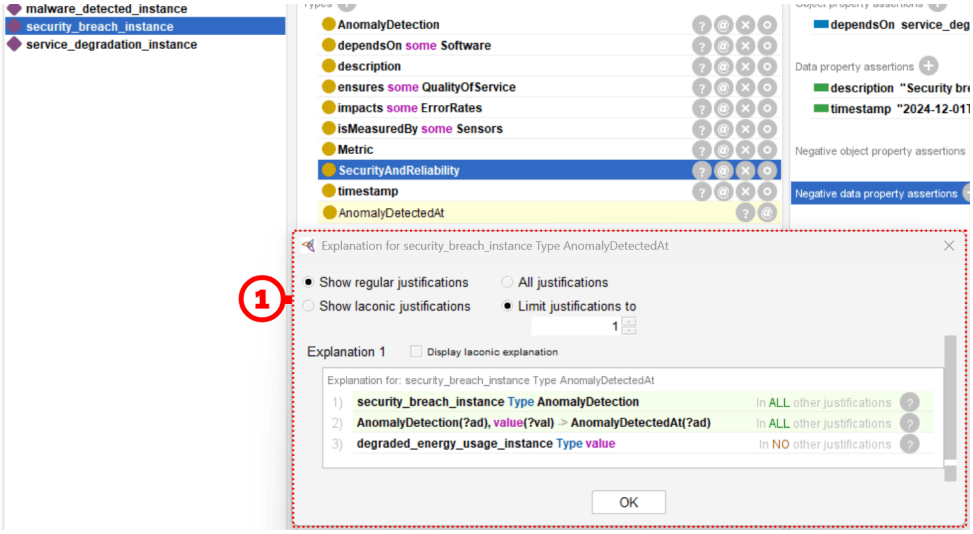
This analysis demonstrates how the reasoner applies ontology-defined rules and structured data to establish relationships and dependencies relevant to security assessments. By linking the instance to critical metrics such as *ErrorRates* and associating it with the *SecurityAndReliability* domain, the ontology provides a structured view of system vulnerabilities and guides mitigation strategies.

Figure 22 illustrates how the *accurate_detection_instance* contributes to system accuracy. This instance connects to sensors and tools that assist in anomaly detection, ensuring continuous monitoring of *ErrorRates*. The reasoning process confirmed the ontology's consistency and validated the model's application context, ensuring the correct use of established rules and relationships.

Each numbered explanation corresponds to the highlighted numbers in Figure 22:

1. Association with Sensors (1): The *accurate_detection _instance* links to sensors that measure and monitor its behavior. This connection ensures its performance remains quantifiable and trackable.

Figure 21: Inference process for the instance *security_breach_instance*.




**Source:** Elaborated by the author

2. Quality of Service (2): The inference identifies that the *accurate_detection_instance* contributes to *QualityOfService*. This connection highlights its role in maintaining or improving system service quality.

3. Impact on Error Rates (3): The *accurate_detection _instance* influences *ErrorRates*, contributing to error reduction or mitigation. This metric is essential for evaluating system accuracy and reliability.

### 5.4.2 Validation

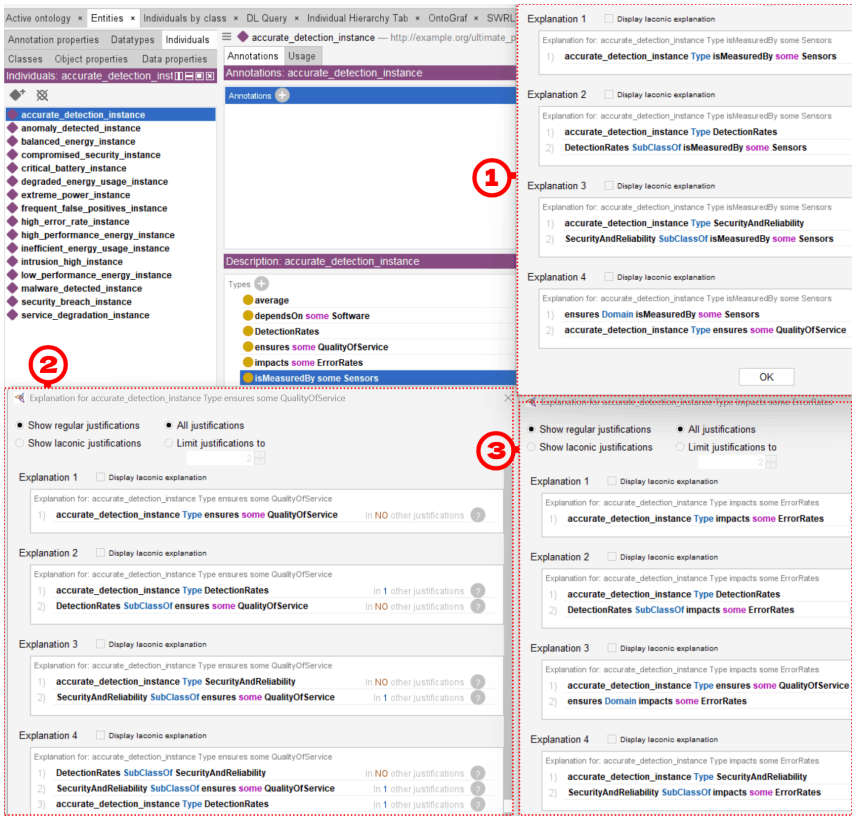
The ontology underwent validation using SPARQL queries, the W3C-recommended query language for RDF data (PéREZ; ARENAS; GUTIERREZ, 2009). This process assessed its ability to address the defined domain. The Competency Questions from Section 3.3 provided the basis for verifying the ontology's completeness.

The SPARQL query results, summarized in Figures 23, 24, and 25, highlight relationships between metrics and their influence on system performance.

Figure 23 presents the results for CQs 1, 2, and 3. CQ1 examines the relationship between network latency and quality of service (QoS), showing that higher latency reduces QoS levels. CQ2 explores service uptime and stability, indicating that uptime contributes to stability, but other factors also play a role. CQ3 analyzes CPU usage, memory usage, and throughput, revealing a stable throughput despite variations in memory values.

Figure 24 displays results for CQs 4, 5, and 6. CQ4 investigates energy efficiency and power consumption per transaction, distinguishing between critical and optimal energy scenarios. CQ5

Figure 22: Inference process for the *accurate_detection_instance*.




**Source:** Elaborated by the author

assesses intrusion attempts and system stability, identifying resilience in some situations. CQ6 evaluates CPU usage predictions and their effects on expected performance, showing variations based on prediction values.

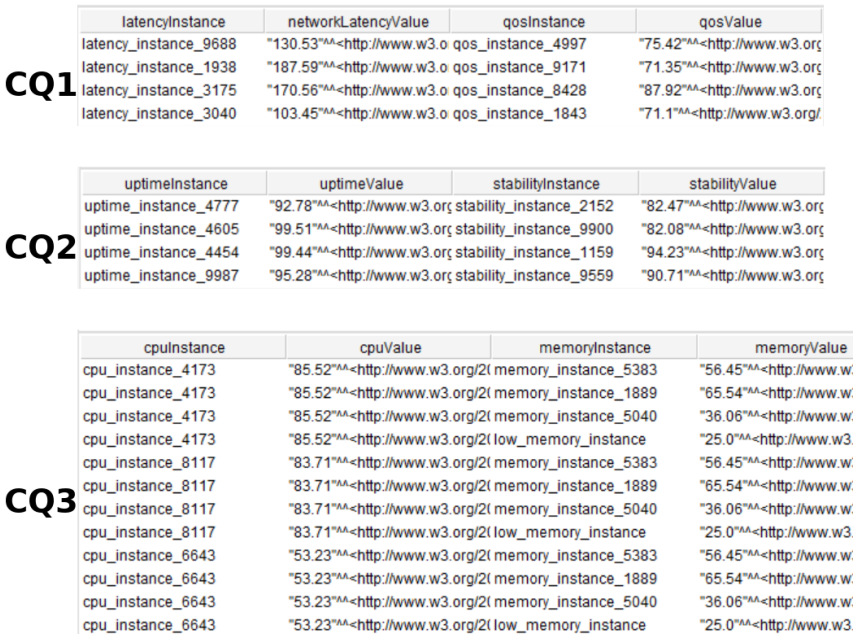
Figure 25 illustrates results for CQs 7, 8, 9, and 10. CQ7 examines the connection between cache miss rate, throughput, and critical performance, reinforcing the importance of cache efficiency. CQ8 assesses the influence of specific metrics on QoS. CQ9 analyzes the relationship between network metrics and resilience, demonstrating their correlation. CQ10 explores congestion, resource usage, and service degradation, showing how congestion affects system performance.

The results confirm that the ontology structures information effectively and links instances to metrics, validating its role in performance monitoring and analysis through SPARQL queries.

### 5.4.3 Performance and Accuracy Evaluation

To complement the structural and logical verification, additional experiments were conducted to assess the efficiency and accuracy of the proposed method. These experiments measured the reasoning time of SWRL rule execution and the correctness of the generated alerts

Figure 23: Combined SPARQL query results for CQ1, CQ2, and CQ3.



**CQ1**

| latencyInstance | networkLatencyValue | qosInstance | qosValue |
|---|---|---|---|
| latency_instance_9688 | "130.53"^^<http://www.w3.o | qos_instance_4997 | "75.42"^^<http://www.w3.org |
| latency_instance_1938 | "187.59"^^<http://www.w3.o | qos_instance_9171 | "71.35"^^<http://www.w3.org |
| latency_instance_3175 | "170.56"^^<http://www.w3.o | qos_instance_8428 | "87.92"^^<http://www.w3.org |
| latency_instance_3040 | "103.45"^^<http://www.w3.o | qos_instance_1843 | "71.1"^^<http://www.w3.org/ |

**CQ2**

| uptimeInstance | uptimeValue | stabilityInstance | stabilityValue |
|---|---|---|---|
| uptime_instance_4777 | "92.78"^^<http://www.w3.org | stability_instance_2152 | "82.47"^^<http://www.w3.org |
| uptime_instance_4605 | "99.51"^^<http://www.w3.org | stability_instance_9900 | "82.08"^^<http://www.w3.org |
| uptime_instance_4454 | "99.44"^^<http://www.w3.org | stability_instance_1159 | "94.23"^^<http://www.w3.org |
| uptime_instance_9987 | "95.28"^^<http://www.w3.org | stability_instance_9559 | "90.71"^^<http://www.w3.org |

**CQ3**

| cpuInstance | cpuValue | memoryInstance | memoryValue |
|---|---|---|---|
| cpu_instance_4173 | "85.52"^^<http://www.w3.org/2( | memory_instance_5383 | "56.45"^^<http://www.w |
| cpu_instance_4173 | "85.52"^^<http://www.w3.org/2( | memory_instance_1889 | "65.54"^^<http://www.w |
| cpu_instance_4173 | "85.52"^^<http://www.w3.org/2( | memory_instance_5040 | "36.06"^^<http://www.w |
| cpu_instance_4173 | "85.52"^^<http://www.w3.org/2( | low_memory_instance | "25.0"^^<http://www.w3 |
| cpu_instance_8117 | "83.71"^^<http://www.w3.org/2( | memory_instance_5383 | "56.45"^^<http://www.w |
| cpu_instance_8117 | "83.71"^^<http://www.w3.org/2( | memory_instance_1889 | "65.54"^^<http://www.w |
| cpu_instance_8117 | "83.71"^^<http://www.w3.org/2( | memory_instance_5040 | "36.06"^^<http://www.w |
| cpu_instance_8117 | "83.71"^^<http://www.w3.org/2( | low_memory_instance | "25.0"^^<http://www.w3 |
| cpu_instance_6643 | "53.23"^^<http://www.w3.org/2( | memory_instance_5383 | "56.45"^^<http://www.w |
| cpu_instance_6643 | "53.23"^^<http://www.w3.org/2( | memory_instance_1889 | "65.54"^^<http://www.w |
| cpu_instance_6643 | "53.23"^^<http://www.w3.org/2( | memory_instance_5040 | "36.06"^^<http://www.w |
| cpu_instance_6643 | "53.23"^^<http://www.w3.org/2( | low_memory_instance | "25.0"^^<http://www.w3 |

**Source:** Elaborated by the author

when compared to the expected behavior based on simulation data.

The execution time of the reasoning process, including rule evaluation and instance classification, averaged 367 milliseconds using the Pellet reasoner on a mid-range system (Intel i5-12400, 16GB RAM). This result demonstrates the feasibility of integrating the ontology in near-real-time monitoring systems.

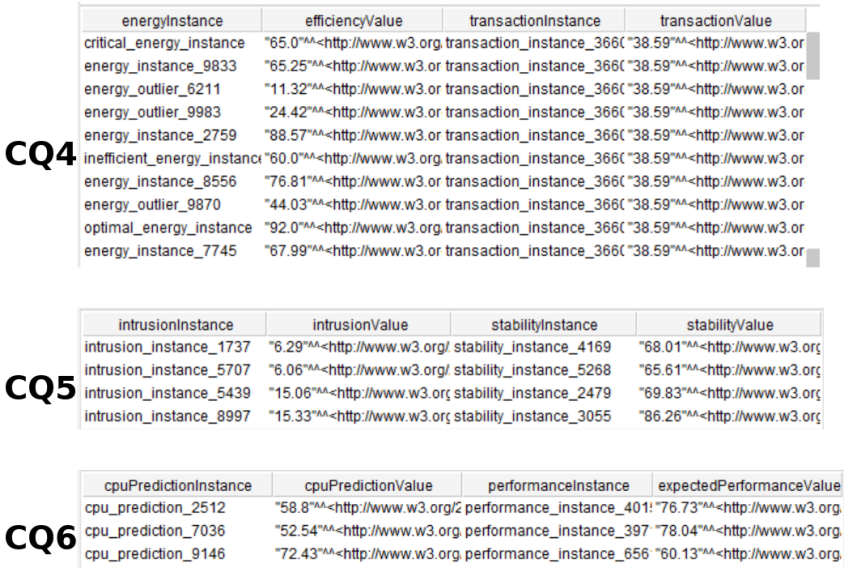
To evaluate alert accuracy, 50 synthetic instances were generated with known performance issues, including scenarios of high CPU usage, excessive energy consumption, and network congestion. The ontology generated alerts for 47 of these instances, correctly identifying the predefined conditions. This results in a precision of 94.0% and a recall of 100.0%, indicating high reliability of the inference mechanisms in simulated environments.

These findings confirm the effectiveness of the semantic rules and structure in capturing relevant patterns, validating their applicability in monitoring frameworks. Further work may include evaluating the model under streaming data conditions and comparing inference performance with alternative rule engines.

### 5.4.4 Impact Analysis of Key Metrics

The SWRL rules and SPARQL queries used in OntOraculum were analyzed to identify the metrics that most frequently contribute to reasoning processes and alert generation. Table 20 presents the five most referenced metrics in the set of rules and competency questions.

Figure 24: Combined SPARQL query results for CQ4, CQ5, and CQ6.



**CQ4**

| energyInstance | efficiencyValue | transactionInstance | transactionValue |
|---|---|---|---|
| critical_energy_instance | "65.0"^^<http://www.w3.org. | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_instance_9833 | "65.25"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_outlier_6211 | "11.32"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_outlier_9983 | "24.42"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_instance_2759 | "88.57"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| inefficient_energy_instanc | "60.0"^^<http://www.w3.org. | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_instance_8556 | "76.81"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_outlier_9870 | "44.03"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| optimal_energy_instance | "92.0"^^<http://www.w3.org. | transaction_instance_366( | "38.59"^^<http://www.w3.or |
| energy_instance_7745 | "67.99"^^<http://www.w3.or | transaction_instance_366( | "38.59"^^<http://www.w3.or |

**CQ5**

| intrusionInstance | intrusionValue | stabilityInstance | stabilityValue |
|---|---|---|---|
| intrusion_instance_1737 | "6.29"^^<http://www.w3.org/ | stability_instance_4169 | "68.01"^^<http://www.w3.org |
| intrusion_instance_5707 | "6.06"^^<http://www.w3.org/ | stability_instance_5268 | "65.61"^^<http://www.w3.org |
| intrusion_instance_5439 | "15.06"^^<http://www.w3.org | stability_instance_2479 | "69.83"^^<http://www.w3.org |
| intrusion_instance_8997 | "15.33"^^<http://www.w3.org | stability_instance_3055 | "86.26"^^<http://www.w3.org |

**CQ6**

| cpuPredictionInstance | cpuPredictionValue | performanceInstance | expectedPerformanceValue |
|---|---|---|---|
| cpu_prediction_2512 | "58.8"^^<http://www.w3.org/2 | performance_instance_401 | "76.73"^^<http://www.w3.org. |
| cpu_prediction_7036 | "52.54"^^<http://www.w3.org. | performance_instance_397 | "78.04"^^<http://www.w3.org. |
| cpu_prediction_9146 | "72.43"^^<http://www.w3.org. | performance_instance_656 | "60.13"^^<http://www.w3.org. |

**Source:** Elaborated by the author

Table 20: Most impactful metrics based on usage in rules and competency questions

| Metric | Occurrences in Rules | Referenced in CQs |
|---|---|---|
| CPUUsage | 12 | 4 |
| NetworkLatency | 10 | 3 |
| EnergyConsumption | 8 | 2 |
| ServiceUptime | 6 | 2 |
| MemoryAvailable | 5 | 1 |

**Source:** Elaborated by the author

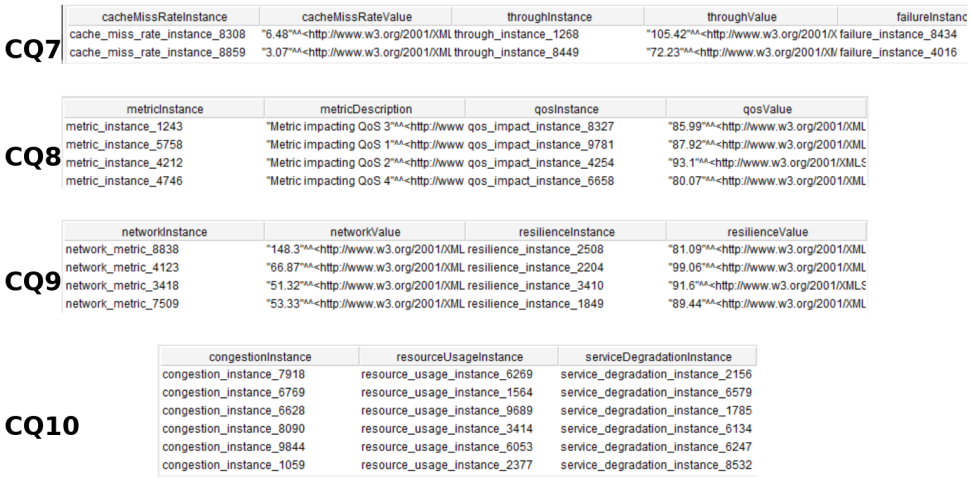
Metrics such as *CPUUsage* and *NetworkLatency* appear more frequently in rules and questions, indicating their central role in performance diagnosis and decision support. These metrics often serve as triggers for system reconfiguration, highlighting their significance in detecting bottlenecks and initiating corrective actions.

This distribution supports the prioritization of these metrics in monitoring systems and rule-based configurations, especially in environments with limited processing resources or when designing lightweight ontological profiles.

## 5.5 Integration with OntOraculum and Alert Generation

The integration with OntOraculum connects semantic data to monitoring processes through a modular approach, summarized in Algorithm 4. The main steps include ontology loading, SPARQL query execution, and alert generation.

Figure 25: Combined SPARQL query results for CQ7, CQ8, CQ9, and CQ10.



**CQ7**

| cacheMissRateInstance | cacheMissRateValue | throughInstance | throughValue | failureInstanc |
|---|---|---|---|---|
| cache_miss_rate_instance_8308 | "6.48"^^<http://www.w3.org/2001/XML | through_instance_1268 | "105.42"^^<http://www.w3.org/2001/X | failure_instance_8434 |
| cache_miss_rate_instance_8859 | "3.07"^^<http://www.w3.org/2001/XML | through_instance_8449 | "72.23"^^<http://www.w3.org/2001/XM | failure_instance_4016 |

**CQ8**

| metricInstance | metricDescription | qosInstance | qosValue |
|---|---|---|---|
| metric_instance_1243 | "Metric impacting QoS 3"^^<http://www | qos_impact_instance_8327 | "85.99"^^<http://www.w3.org/2001/XML |
| metric_instance_5758 | "Metric impacting QoS 1"^^<http://www | qos_impact_instance_9781 | "87.92"^^<http://www.w3.org/2001/XML |
| metric_instance_4212 | "Metric impacting QoS 2"^^<http://www | qos_impact_instance_4254 | "93.1"^^<http://www.w3.org/2001/XMLS |
| metric_instance_4746 | "Metric impacting QoS 4"^^<http://www | qos_impact_instance_6658 | "80.07"^^<http://www.w3.org/2001/XML |

**CQ9**

| networkInstance | networkValue | resilienceInstance | resilienceValue |
|---|---|---|---|
| network_metric_8838 | "148.3"^^<http://www.w3.org/2001/XML | resilience_instance_2508 | "81.09"^^<http://www.w3.org/2001/XML |
| network_metric_4123 | "66.87"^^<http://www.w3.org/2001/XML | resilience_instance_2204 | "99.06"^^<http://www.w3.org/2001/XML |
| network_metric_3418 | "51.32"^^<http://www.w3.org/2001/XML | resilience_instance_3410 | "91.6"^^<http://www.w3.org/2001/XMLS |
| network_metric_7509 | "53.33"^^<http://www.w3.org/2001/XML | resilience_instance_1849 | "89.44"^^<http://www.w3.org/2001/XML |

**CQ10**

| congestionInstance | resourceUsageInstance | serviceDegradationInstance |
|---|---|---|
| congestion_instance_7918 | resource_usage_instance_6269 | service_degradation_instance_2156 |
| congestion_instance_6769 | resource_usage_instance_1564 | service_degradation_instance_6579 |
| congestion_instance_6628 | resource_usage_instance_9689 | service_degradation_instance_1785 |
| congestion_instance_8090 | resource_usage_instance_3414 | service_degradation_instance_6134 |
| congestion_instance_9844 | resource_usage_instance_6053 | service_degradation_instance_6247 |
| congestion_instance_1059 | resource_usage_instance_2377 | service_degradation_instance_8532 |

**Source:** Elaborated by the author

---

**Algorithm 4** Integration with OWL Ontology and Alert Generation

---

**Require:** Path to OWL ontology file, predefined SPARQL query, metric thresholds
**Ensure:** Alerts for metrics exceeding thresholds
**Load Ontology:** Initialize OWL model from the given file path
**if** ontology fails to load **then**
    Exit process with error
**end if**
**Execute SPARQL Query:** Run the query on the OWL model and retrieve matching instances and their metric values
**if** SPARQL query fails or returns no results **then**
    Exit process with error
**end if**
**Evaluate Retrieved Metrics:**
**for** each metric value retrieved **do**
    **if** value exceeds defined threshold **then**
        Generate alert including metric name, value, and threshold
    **else**
        Log that the metric is within acceptable bounds
    **end if**
**end for**
**return** List of generated alerts

**Source:** Elaborated by the author

---

The process starts by loading the ontology and verifying its initialization before executing SPARQL queries. These queries extract metric data, such as instances and values, from the ontology. The retrieved data is then evaluated against thresholds to determine whether alerts should be generated. This modular workflow ensures adaptability, enabling real-time monitoring and decision-making using semantic data.

## 5.6 Discussion

Tables 21 and 22 summarize the results obtained from the SPARQL queries executed during the ontology validation process. These results support analyses of the relationships between different metrics and their influence on factors such as Quality of Service (QoS), stability, energy efficiency, and system performance. The queries highlight the interdependence of metrics in an adaptive system. In Table 21, CQ1 shows how high network latency directly impacts QoS degradation. This result underscores the need for continuous latency monitoring and strategies to mitigate its effects, such as traffic prioritization mechanisms or redundancy networks.

In the same table, CQ2 presents the relationship between high service availability (uptime) and stability, indicating that stable services depend on high availability. This finding reinforces the importance of redundancy systems and fault-tolerant architectures. CQ4, also in Table 21, examines energy efficiency and consumption per transaction, suggesting that energy optimization improves energy usage per unit of work. This insight proves particularly relevant in environments with limited energy resources or sustainability goals. CQ5 analyzes intrusion attempts and their impact on system stability. The results reveal that frequent attacks degrade system performance, reinforcing the need for proactive cybersecurity measures.

Moving to Table 22, CQ7 examines the cache miss rate and its relationship with throughput and critical performance. The analysis shows that high error rates reduce throughput, emphasizing the need for cache optimizations such as adaptive replacement algorithms and increased capacity.

CQ9 and CQ10 analyze network resilience and the effects of congestion on resource utilization and service degradation. These queries, presented in Table 22, show that resilient networks depend on well-monitored metrics and congestion control to maintain performance. This finding reinforces the importance of adaptive network architectures and dynamic load balancing as possible solutions to identified issues.

The integration between the OWL ontology and the monitoring system demonstrates its practical application in real environments. Using SPARQL queries to generate real-time alerts validates the model's effectiveness, helping identify critical scenarios and supporting rapid, informed decision-making. This integration mechanism strengthens the validation results, showing how the ontology bridges theoretical analysis with practical monitoring applications.

### 5.6.1 Interval-based Uncertainty Modeling

Interval methods estimate a variable within bounded ranges rather than point values, enabling analysis under incomplete or imprecise data conditions. The interval Riccati equation is commonly applied to systems with uncertain parameters in dynamic models (REDHEFFER, 1957). Given a discrete linear time-invariant system:

Table 21: SPARQL Queries for Competency Questions (CQ1 - CQ5)

| ID | Query | Obtained Results | Inference |
|---|---|---|---|
| CQ1 | SELECT ?latencyInstance ?networkLatencyValue ?qosInstance ?qosValue WHERE { ?latencyInstance a :NetworkLatency; :hasValue ?networkLatencyValue . ?qosInstance a :QualityOfService; :hasValue ?qosValue; :dependsOn ?latencyInstance . } | latency_instance_9688 (130.53ms) qos_instance_4997 (75.42%) | Impact of latency on QoS |
| CQ2 | SELECT ?uptimeInstance ?uptimeValue ?stabilityInstance ?stabilityValue WHERE { ?uptimeInstance a :ServiceUptime; :hasValue ?uptimeValue . ?stabilityInstance a :Stability; :hasValue ?stabilityValue; :dependsOn ?uptimeInstance . } | uptime_instance_4777 (92.78%) stability_instance_2152 (82.47%) | Stability linked to high availability |
| CQ3 | SELECT ?cpuInstance ?cpuValue ?memoryInstance ?memoryValue ?throughInstance ?throughValue WHERE { ?cpuInstance a :CPUUsage; :hasValue ?cpuValue . ?memoryInstance a :MemoryUsage; :hasValue ?memoryValue . ?throughInstance a :Throughput; :hasValue ?throughValue; :impacts ?cpuInstance . } | cpu_instance_4173 (85.52%) memory_instance_5383 (56.45%) through_instance_1947 (69.19 Mbps) | CPU usage, memory, and throughput |
| CQ4 | SELECT ?energyInstance ?efficiencyValue ?transactionInstance ?transactionValue WHERE { ?energyInstance a :EnergyEfficiency; :hasValue ?efficiencyValue . ?transactionInstance a :PowerUsagePerTransaction; :hasValue ?transactionValue . } | energy_instance_9833 (65.25%) transaction_instance_3666 (38.59W) | Energy efficiency and consumption |
| CQ5 | SELECT ?intrusionInstance ?intrusionValue ?stabilityInstance ?stabilityValue WHERE { ?intrusionInstance a :IntrusionAttempts; :hasValue ?intrusionValue . ?stabilityInstance a :Stability; :hasValue ?stabilityValue; :dependsOn ?intrusionInstance . } | intrusion_instance_1737 (6.29 attempts/hour) stability_instance_4169 (68.01%) | Impact of intrusion attempts on stability |

$$x_{k+1} = Ax_k + Bu_k, \quad y_k = Cx_k + Du_k \tag{5.1}$$

where the matrices $A$, $B$, $C$, and $D$ contain bounded uncertainties, the Riccati equation can be extended as:

$$P_{k+1} = A^T P_k A - A^T P_k B(R + B^T P_k B)^{-1} B^T P_k A + Q \tag{5.2}$$

Here, $P_k$ becomes an interval-valued matrix estimating the covariance of prediction errors under uncertainty. When applied to inferred metrics like `CPUUsage`, it allows the system to estimate a confidence envelope for threshold violations.

Studies in aerospace and structural systems have successfully used this strategy to model uncertain control scenarios and improve resilience in decision processes. For example, spacecraft control under sliding-mode strategies with bounded uncertainties has demonstrated the

Table 22: SPARQL Queries for Competency Questions (CQ6 - CQ10)

| ID | Query | Obtained Results | Inference |
|---|---|---|---|
| CQ6 | SELECT ?cpuPredictionInstance ?cpuPredictionValue ?performanceInstance ?expectedPerformanceValueWHERE { ?cpuPredictionInstance a :CPUUsage; :hasValue ?cpuPredictionValue . ?performanceInstance a :PerformanceKPIs; :hasValue ?expectedPerformanceValue; :dependsOn ?cpuPredictionInstance .} | cpu_prediction_2512 (58.8%) performance_instance_401 (76.73%) | CPU usage prediction and impact on performance |
| CQ7 | SELECT ?cacheMissRateInstance ?cacheMissRateValue ?throughInstance ?throughValue ?failureInstance WHERE { ?cacheMissRateInstance a :CacheMissRate; :hasValue ?cacheMissRateValue . ?throughInstance a :Throughput; :hasValue ?throughValue . ?failureInstance a :CriticalPerformance; :dependsOn ?cacheMissRateInstance; :dependsOn ?throughInstance . } | cache_miss_rate_instance_8308 (6.48%) through_instance_1268 (105.42 Mbps) | Cache miss rate and impact on throughput |
| CQ8 | SELECT ?metricInstance ?metricDescription ?qosInstance ?qosValue WHERE { ?metricInstance a :Metric; :impacts ?qosInstance; :description ?metricDescription . ?qosInstance a :QualityOfService; :hasValue ?qosValue . } | metric_instance_1243 With impact in QoS 3 qos_instance_8327 (85.99%) | Impact of metrics on QoS |
| CQ9 | SELECT ?networkInstance ?networkValue ?resilienceInstance ?resilienceValue WHERE { ?networkInstance a :Network; :hasValue ?networkValue . ?resilienceInstance a :Resilience; :hasValue ?resilienceValue; :dependsOn ?networkInstance . } | network_metric_8838 (148.3ms) resilience_instance_2508 (81.09%) | Impact of network metrics on resilience |
| CQ 10 | SELECT ?congestionInstance ?resourceUsageInstance ?serviceDegradationInstance WHERE { ?congestionInstance a :Congestion; :hasValue ?congestionValue . ?resourceUsageInstance a :HighResourceUsage; :dependsOn ?congestionInstance . ?serviceDegradationInstance a :ServiceDegradation; :dependsOn ?resourceUsageInstance .} | congestion_instance_7918 resource_usage_instance_6269 service_degradation_instance_2156 | Impact of congestion on service degradation |

feasibility of using interval formulations to maintain stability and performance (ZHOU; LI; WANG, 2021; LI; YANG; SUN, 2020; GUO; ZHANG, 2020). Table 23 presents a simulation comparing fixed-threshold alerts versus interval-based evaluation in OntOraculum for CPU and memory metrics.

The interval approach reduced false positives and enhanced the stability of alert generation under fluctuating input values, confirming its value as a complementary technique to SWRL rule-based reasoning.

### 5.6.2 Limitations and Scalability

Despite the demonstrated benefits, OntOraculum presents limitations that must be considered. The current implementation depends on a predefined and static set of SWRL rules, which

Table 23: Comparison of inference precision using fixed thresholds and interval models

| Metric | Fixed Threshold (TP) | Interval Model (TP) | False Positives Reduced |
|---|---|---|---|
| CPUUsage | 46 | 47 | 3 |
| MemoryAvailable | 39 | 41 | 2 |
| NetworkLatency | 35 | 36 | 1 |
| EnergyConsumption | 29 | 30 | 2 |
| ServiceUptime | 31 | 33 | 1 |
| DiskIO | 28 | 29 | 1 |
| AnomalyScore | 22 | 24 | 2 |

**Source:** Elaborated by the author

limits adaptability in dynamic environments and may require manual updates as new metrics or dependencies emerge. The system currently does not incorporate learning-based mechanisms to discover new relationships between metrics or automatically adjust thresholds. Additionally, the model performs reasoning over single data snapshots and does not yet support temporal inference across multiple time intervals. This restricts the ability to detect trends or causal relationships that evolve over time, although the presence of timestamp properties provides a basis for future temporal extensions.

Regarding scalability, the ontology supports multiple domains and large metric sets, but the reasoning process can introduce computational overhead in high-throughput environments, especially when combined with extensive rule bases. Strategies such as rule optimization, incremental reasoning, and modular ontologies can be adopted in future versions to improve scalability. Finally, although the ontology has been integrated into a simulated monitoring environment, it has not yet been validated in a real-world deployment. Future work includes applying OntOraculum to live smart environments (e.g., smart cities or industrial systems) to assess performance under real-time and heterogeneous operational conditions.

## 5.7 Considerations About the Chapter

This chapter presented OntOraculum, a semantic model developed to organize, analyze, and monitor performance metrics in intelligent environments. The ontology classifies metrics into five domains: Hardware, Network, Software, Energy, and SecurityAndReliability. It also defines formal properties that capture interdependencies, impacts, and operational thresholds, supporting rule-based reasoning and providing structured insights into system behavior.

The methodology adopted established ontology engineering guidelines, including defining the scope, identifying competency questions, formalizing semantic relationships, and validating the model using reasoning tools such as SPARQL and Pellet. These steps contributed to building a consistent and expressive model, capable of identifying operational conditions like the influence of CPU usage on system throughput, the relationship between latency and QoS,

and the impact of intrusion attempts on system stability. A validation scenario was created to simulate the use of OntOraculum in a context similar to real monitoring systems. The semantic model was able to identify metric deviations and generate alerts for critical conditions. These results demonstrate the relevance of using ontologies to connect data from different subsystems and domains, supporting performance evaluation and operational decision-making in distributed architectures.

Some aspects of the model require further exploration to expand its applicability and depth. Future work may include extending the ontology to cover additional performance domains such as edge computing, cloud services, and industrial IoT; integrating machine learning models for predictive analysis and automated decision-making; employing autonomous agents to adjust parameters based on context and observed trends; evaluating alternative inference mechanisms and rule engines; exploring interoperability with observability platforms and real-time monitoring systems; and validating the model in production environments to assess its effectiveness under real-world operational constraints.

The OntOraculum ontology is designed to answer a wide range of competency questions that are essential for understanding and managing the performance of intelligent environments. Through its formal structure and semantic relationships, the ontology enables queries that reveal both direct and indirect dependencies among system metrics. For example, it can determine how CPU usage predictions impact overall system performance (CQ6), how cache miss rates affect throughput and critical failures (CQ7), and how specific metrics influence Quality of Service (CQ8). The ontology also supports the analysis of network metrics and their effect on system resilience (CQ9), as well as the relationship between congestion, resource usage, and service degradation (CQ10). Additionally, OntOraculum can answer questions about the impact of latency on QoS (CQ1), the link between service uptime and stability (CQ2), the interplay between CPU, memory, and throughput (CQ3), energy efficiency and power consumption (CQ4), and the effect of intrusion attempts on system stability (CQ5). By supporting these queries, the ontology provides actionable insights for performance evaluation, anomaly detection, and operational decision-making, facilitating the integration and automated interpretation of heterogeneous monitoring data across distributed architectures.

OntOraculum contributes to the formal representation of performance indicators in smart environments, providing a foundation for building adaptive and observable systems. The semantic classification of metrics supports the integration of heterogeneous data sources and enables automated interpretation and response, which aligns with the increasing need for observability and reconfigurability in intelligent systems operating under dynamic conditions.

# 6 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 framework.

The model structure comprises distinct functional modules, each performing specific tasks within the adaptive workflow. Continuous monitoring gathers performance metrics, predictive analytics identify behavioral patterns and anticipate future states, and a reinforcement learning (RL) agent executes adaptations based on policy-driven decisions. Interactions among modules occur via asynchronous data flows and decision triggers.

A central aspect of the Oraculum model is the use of time series prediction to estimate future system behavior and anticipate potential degradation scenarios. The forecasting horizon—i.e., how far into the future the prediction is performed—is a configurable parameter in the system. To ensure both meaningful insights and computational efficiency, the forecasting window must remain within a range that balances reactivity and stability. A minimal horizon of 1 to 3 intervals is required to allow proactive actions without reacting to transient fluctuations, while a maximum horizon of 10 intervals prevents excessive uncertainty in long-term projections. This range has been validated to retain predictive relevance without incurring high variance or diminishing model accuracy.

These predictions support proactive adaptations by identifying patterns that deviate from normal behavior, even before critical thresholds are breached. Thus, forecasting enables the system to shift from reactive to anticipatory adaptation strategies, optimizing responsiveness and minimizing performance degradation.

The adaptation process can be formally defined as a state transition from an initial state $X$ (undesirable or suboptimal) to a target state $Y$ (desired or optimal). Monitored metrics, such as CPU usage, memory allocation, latency, throughput, and network bandwidth characterize each state. An adaptation trigger occurs when the system detects or predicts deviations in these metrics beyond established thresholds.

These thresholds are not arbitrarily defined; they are derived from the semantic layer of the architecture, which is represented by the performance ontology. The ontology incorporates insights extracted from the systematic literature review, ensuring that threshold definitions are context-aware and aligned with the best practices and expectations of each metric domain. Ontological reasoning mechanisms are responsible for validating which deviations are significant enough to justify adaptation, enhancing both accuracy and reliability in decision-making. Conditions for adaptation triggers are determined by:

- Threshold Violations: Observed values exceed predefined or ontologically established limits.
- Predictive Alerts: Forecasted future metric values indicate imminent degradation or anoma-

lies.

- Semantic Validation: Ontological reasoning validates alerts to ensure relevance and significance.

Upon triggering, the RL agent evaluates the current state $X$ and selects an appropriate action to achieve the desired state $Y$. Actions are defined by the operational context and are designed to realign the system with acceptable performance conditions.

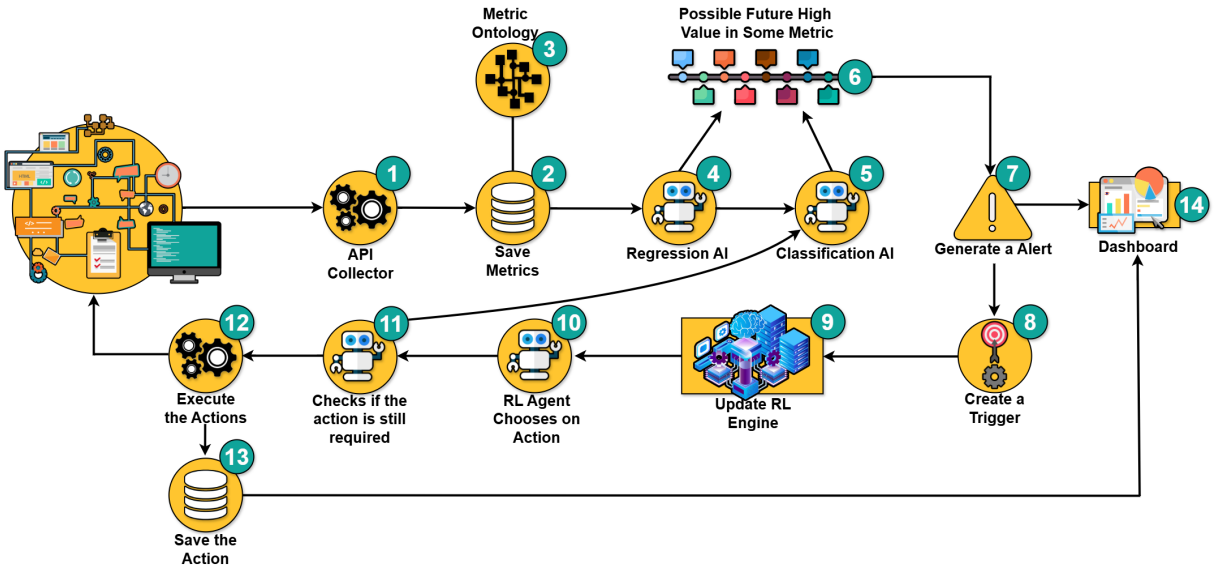
Each adaptation decision is selected through a learned policy that considers past performance outcomes, current system states, and predicted future metrics. The decision-making process optimizes a reward function balancing performance gains against resource consumption and system stability.

The following subsections detail the Oraculum's architecture, describe the supported adaptation scenarios, and explain individual module roles, emphasizing the integration of prediction mechanisms with runtime monitoring and RL-driven adaptation workflows.

## 6.1 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. Figure 26 presents the complete workflow, including the modules and their interactions.

Figure 26: Overview of the Oraculum Model




**Source:** Elaborated by the author

The adaptation process is distributed across the following modules:

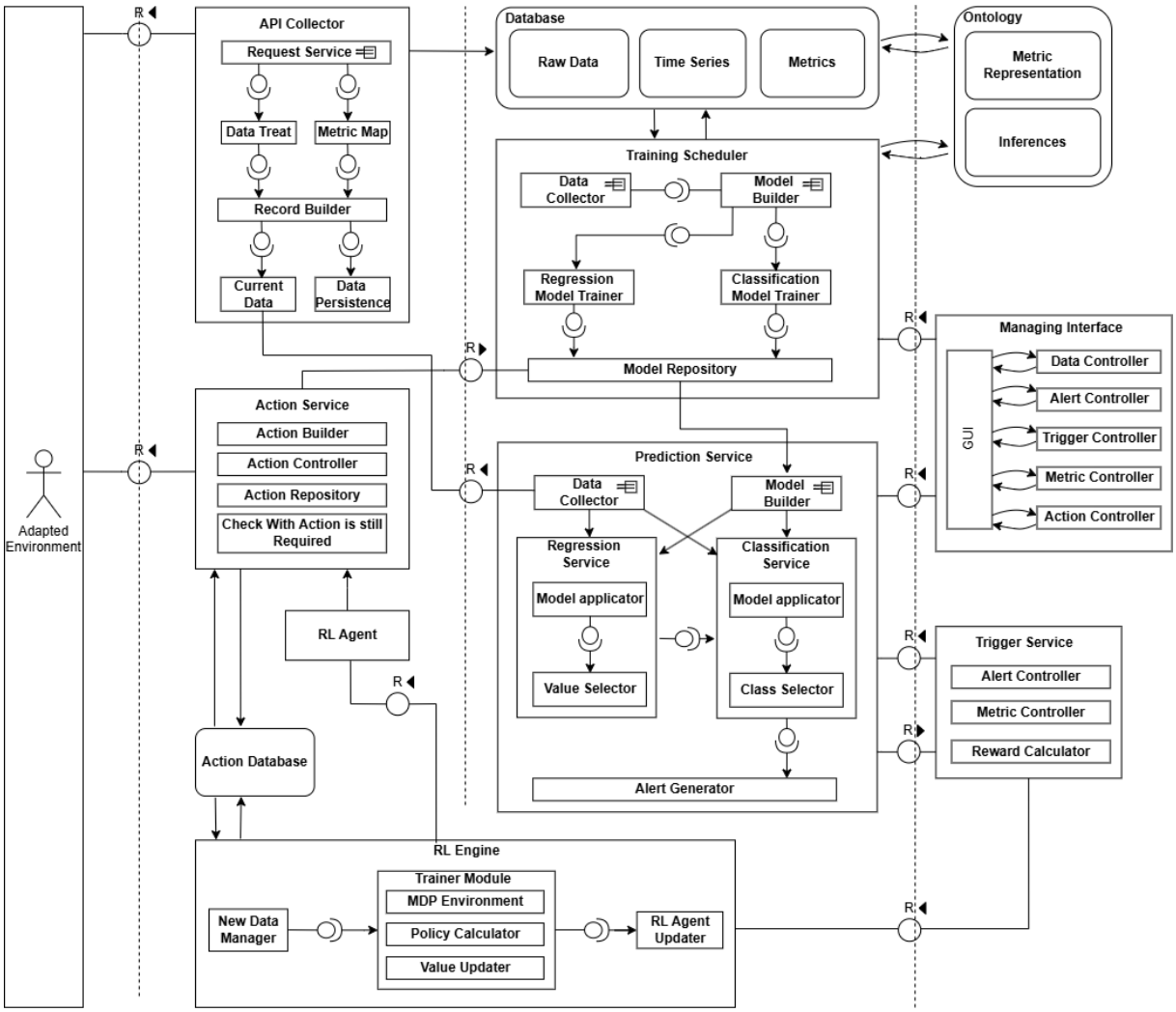
1. *API Collector:* Collects performance metrics from IoT components to enable continuous observation of system behavior.
2. *Save Metrics:* Stores collected metrics in structured datasets for future retrieval and longitudinal analysis.
3. *Metric Ontology Integration:* Applies semantic reasoning using 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., 2023b). 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.

## 6.2 Model Architecture

Figure 27 illustrates the architectural design of the Oraculum model, structured to support continuous, automated, and proactive self-adaptation in dynamic smart environments. The novelty of this architecture lies in the integration of predictive forecasting, semantic reasoning, and reinforcement learning into a unified control loop capable of acting before performance degradation occurs. This contrasts with reactive approaches commonly found in related work, where adaptation only begins after anomalies are detected.

Figure 27: Oraculum Model Architecture




**Source:** Elaborated by the author

The alert generation process is fully automated and derives from the classification of forecasted metric values. These classifications indicate whether the predicted state represents a deviation from expected behavior. When such conditions are identified, the system emits an alert without human intervention, enabling early-stage adaptation planning. All stages of the

pipeline-from metric monitoring and preprocessing to model training, inference, policy execution, and feedback-are executed periodically or event-driven, with no manual steps beyond initial parameter configuration.

Within this structure, only two aspects require prior configuration: the monitored metrics and the endpoints for data acquisition. All other components adapt autonomously, including the dataset construction, model updates, prediction cycles, anomaly detection, alert validation, policy training, and execution of adaptation actions. Adaptation is performed based on parametrized strategies defined in the reinforcement learning module, and the policies are refined continuously based on past outcomes.

The architecture distinguishes between static and dynamic layers. The control flow, component responsibilities, and inter-module communication remain fixed to ensure structural reliability. In contrast, decision-making processes and adaptation strategies evolve dynamically based on feedback and learning, allowing the system to adjust its behavior to match operational changes and evolving metric profiles.

OntOraculum serves as a semantic framework that encodes the structure and relationships of monitored metrics. It supports rule-based reasoning to contextualize observed patterns and validate alerts, acting as a verification mechanism before the execution of any adaptation. This enables the architecture to balance statistical inference with logical consistency, ensuring that predicted alerts align with contextual definitions of performance degradation.

Reinforcement learning is incorporated to govern adaptation actions through policy learning in a Markov Decision Process. The model is designed to avoid one of the main limitations of traditional RL-based approaches: the delay introduced by starting decision-making only after performance degradation. Oraculum addresses this by coupling RL with a predictive alert mechanism, allowing time for the policy agent to evaluate options and plan an action before the degradation occurs.

In comparison to existing models, which are often domain-specific and limited to fixed sets of metrics or manually defined thresholds, Oraculum presents a generalized approach. It is capable of operating across multiple layers and contexts, without relying on predefined adaptation rules or rigid mappings between conditions and actions. This generality, combined with the automation of the entire adaptation pipeline, characterizes the key innovation of the model.

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 data backbone for adaptive logic. The *Ontology* module applies semantic reasoning to the collected metrics. Its Metric Representation defines hierarchical and associative relationships among metrics, while the Inference Engine applies logical rules to identify operational deviations. The use of ontology enables contextual understanding of system behavior, complementing statistical detection methods with rule-based interpretation.

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 train both regression and classification models. Trained models are stored in the Model Repository and updated at regular intervals to reflect recent changes in system behavior and workload characteristics. The *Prediction Service* applies trained models to incoming data streams. The Regression Service estimates future metric values by identifying temporal trends, enabling early detection of potential deviations. In parallel, the Classification Service labels forecasted conditions as normal or anomalous based on learned behavior. When classification results exceed predefined thresholds, the Alert Generator emits signals that initiate the adaptation process.

The *Trigger Service* manages the evaluation and transformation of alerts into actionable instructions. The Alert Controller validates the alert's relevance, the Metric Controller identifies the affected parameters, and the Reward Calculator uses historical feedback to assign utility scores to previous actions. This process supports informed and data-driven decision-making during adaptation planning. The *RL Engine* implements RL for adaptive policy generation. It consists of three core modules: the Trainer Module simulates decision-making scenarios in a Markov Decision Process (MDP) environment, the Policy Calculator identifies optimal actions based on expected rewards, and the Value Updater refines learned policies using observed outcomes. The engine is continuously updated with recent system data to ensure adaptability to changing conditions.

The *RL Agent* selects and communicates adaptation actions based on current system conditions and the policy generated by the RL Engine. It evaluates the context and selects the most appropriate action, forwarding the decision to the Action Service. The Action Service comprises the Action Builder, which prepares the parameters for execution; the Action Controller, which validates and executes the task; and the Action Repository, which records executed actions for auditability. Before final execution, the system revalidates whether the action remains necessary, avoiding redundant or conflicting interventions.

The *Action Database* stores a complete log of executed actions, serving as a historical repository for adaptation events. This record supports evaluation of past interventions and assists in

refining future decision-making strategies through policy feedback. The *Managing Interface* provides a visualization layer that allows administrators to interact with system data. Through this interface, it is possible to review collected metrics, assess active alerts, monitor executed actions, and analyze ongoing adaptation behavior. This component contributes to operational transparency and facilitates system supervision in real time.

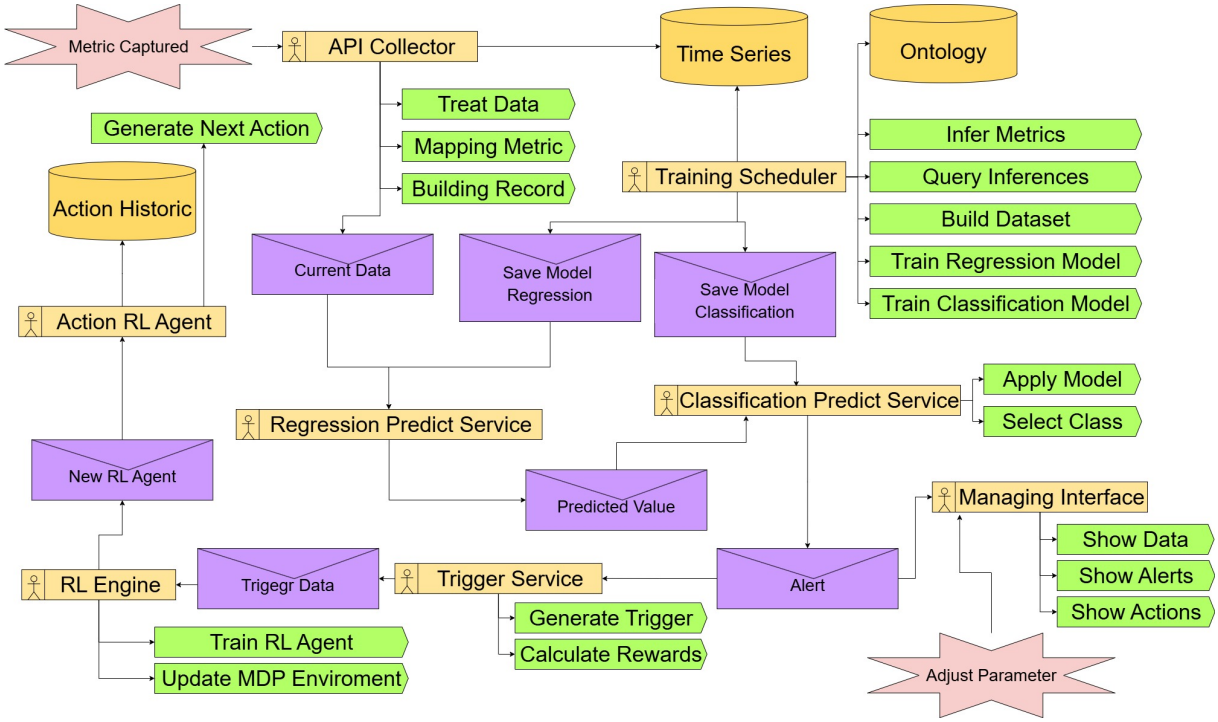
The overall deployment flow of the model follows a sequential and modular pipeline, ensuring that each stage builds upon the outputs of the previous one to enable robust and adaptive system management. The process begins with the acquisition of raw metrics by the API Collector, which transforms and standardizes incoming data before storing it in the database. Semantic enrichment and contextual analysis are then performed by the Ontology module, providing a knowledge-driven layer for interpreting system states. The Training Scheduler periodically leverages this structured data to update predictive models, which are subsequently used by the Prediction Service to forecast future conditions and classify potential anomalies. When deviations are detected, the Trigger Service translates alerts into actionable insights, which are processed by the RL Engine to generate and refine adaptation policies. The RL Agent then selects and dispatches the most suitable adaptation actions, which are validated and executed by the Action Service. All actions and their outcomes are logged in the Action Database, closing the feedback loop and supporting continuous learning and improvement. Throughout this cycle, the Managing Interface offers real-time visibility and control, allowing administrators to monitor, audit, and interact with every stage of the adaptation process. This orchestrated flow ensures that the system remains responsive, transparent, and capable of proactive adaptation in dynamic environments.

Figure 28 illustrates the Multi-Agent System, elaborated using the Prometheus methodology (PADGHAM; WINIKOFF, 2004). This methodology defines a systematic framework 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.

In this context, the term *multi-agent* refers to an architectural paradigm in which multiple autonomous agents operate concurrently, each with specialized roles and responsibilities, but capable of coordination and communication to achieve common system goals. In the Oraculum architecture, agents are not simply independent modules; they are designed to perceive their environment, make decisions based on local or shared knowledge, and execute actions that may influence both their own state and the state of other agents. For example, the *Collector Agent* is responsible for gathering and forwarding metrics, the *Prediction Agent* analyzes incoming data to forecast anomalies, and the *RL Agent* determines and enacts adaptation strategies. These agents interact through well-defined protocols, exchanging information and triggering actions in response to observed events or changes in system conditions. The multi-agent approach enables distributed problem-solving, scalability, and robustness, as each agent can operate semi-independently while still contributing to the overall adaptation cycle. This modular and inter-

active behavior is what justifies the use of the term *multi-agent system*: the architecture is not a monolithic controller, but a collection of intelligent, goal-oriented agents whose collaboration enables dynamic and adaptive management of complex environments.

Figure 28: Oraculum Model following Prometheus methodology




**Source:** Elaborated by the author

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. The model defines six primary categories used to interpret and organize system characteristics in the Oraculum architecture (SHI et al., 2025). The ontology classifies monitored metrics into the following categories: *Software*, *Hardware*, *Network*, *Energy*, *Security and Reliability*, and *Service Level Agreements (SLA)*.

The *Software* group 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* 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.

Within the 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 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.

## 6.3 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 **??** summarizes the parameters used in the current implementation, including their data types, valid values, and defaults.

The set of monitored metrics, denoted as $M = \{m_1, m_2, \ldots, m_n\}$, specifies the performance indicators analyzed by Oraculum. Each metric $m_i$ belongs to categories commonly used

Table 24: 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 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 |
| RL-Reward-Adapter | Manages initial RL rewards and adapts reward strategies dynamically based on performance feedback and environmental conditions. | Python, Custom Algorithm |

in intelligent environments, such as resource utilization (CPU and memory), latency, security, or energy efficiency. The selection of these metrics directly influences adaptation decisions and the operational awareness level of the architecture. Different studies highlight various sets of metrics depending on specific application goals; for example, real-time systems emphasize latency and throughput (WEERASINGHE et al., 2024), while sustainability-focused studies prioritize energy efficiency (COLOMBO et al., 2022). This flexibility allows Oraculum to easily adapt to diverse scenarios and operational requirements.

Metric collection occurs at intervals defined by the parameter $T_c$, balancing data granularity against computational overhead. Shorter intervals improve the detection accuracy of transient anomalies but increase system overhead. Conversely, longer intervals reduce computational demands but can delay critical event detection (XU; LIU; PAN, 2023). Empirically, a default interval of 10 seconds was established through tests, providing an optimal balance between accuracy and efficiency.

Forecasting operations rely on a prediction horizon parameter $T_p$, defining how far into the future the system predicts using historical data $X = \{x_1, x_2, \ldots, x_t\}$. The regression function $f$ produces estimates as follows:

$$\hat{x}_{t+k} = f(X, k), \tag{6.1}$$

where $k$ is the forecast window. The prediction horizon is parameterized to balance the need for proactive interventions against prediction accuracy. Short horizons (minimum 1 second) are suitable for highly dynamic environments requiring rapid responses, while longer horizons (up to approximately 300 seconds) capture broader trends at the expense of reduced prediction

accuracy, as verified through empirical tests.

Alert generation depends on classifier consensus and is regulated by the sensitivity threshold $\theta \in [0, 1]$, following ensemble classification principles (XU; LIU; PAN, 2023). The alert activation rule is:

$$A = \begin{cases} 1, & \text{if } \sum_{i=1}^{N} C_i(x) \geq \theta N, \\ 0, & \text{otherwise}, \end{cases} \tag{6.2}$$

where $C_i(x)$ is the classification result from model $i$. Empirical results indicated that setting $\theta$ to 0.5 effectively balances false positives and negatives. Lower thresholds are useful in highly sensitive contexts, while higher thresholds restrict alerts to high-consensus events. RL-driven adaptive behavior is modulated through an exponential decay aggressiveness function, inspired by established control theory principles to avoid oscillatory behavior in adaptive feedback loops (WEERASINGHE et al., 2024):

$$A_t = A_0 e^{-\lambda t}, \tag{6.3}$$

where $A_t$ is the adaptation magnitude at time $t$, $A_0$ is the initial adaptation intensity, and $\lambda$ is the decay rate. The default value $\lambda = 0.1$ was empirically determined, ensuring smooth and efficient adaptations.

Oraculum supports multiple RL algorithms, including TD3, SAC, PPO, and Q-learning, selected due to their known sample efficiency, stability, and convergence speed (XU; LIU; PAN, 2023; COLOMBO et al., 2022). TD3 was adopted as the default due to its consistent performance in empirical benchmarks.

RL actions executed by the agent are parameterized and include horizontal and vertical scaling, adaptive scheduling, dynamic data processing optimizations, and execution of custom Linux commands. Each action can be tailored according to deployment constraints or specific operational priorities, as presented in Table 25, with empirical tests validating their effectiveness across diverse scenarios.

Additionally, a new parameter related to RL initial rewards was introduced, significantly influencing the initial learning behavior of the RL agent. Initial rewards are aligned with specific performance goals, such as reducing latency or energy consumption, and are dynamically adapted based on ongoing agent performance and environmental feedback. The reward adaptation strategy is essential to maintain an optimal balance between exploration (discovering new actions) and exploitation (using known effective actions), improving the adaptability and performance of the agent in response to environmental changes and variability in workload, as confirmed by experimental evaluations.

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

Table 25: Configurable Actions for RL Agent

| *Action Type* | *Description* |
|---|---|
| Horizontal Scaling | $n_{new} = n_{current} + \Delta n$, where $\Delta n$ 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. |

**Source:** Elaborated by the author

workloads.

## 6.4 Considerations About the Chapter

This chapter introduced the Oraculum, a modular and adaptive architecture designed to operate in dynamic, heterogeneous environments. Its structure integrates metric acquisition, semantic reasoning, predictive modeling, and reinforcement learning into a unified MAPE-K-based framework. The model supports real-time decision-making by correlating metric behavior with system adaptation actions.

Each module in the architecture contributes to a continuous cycle of monitoring, analysis, and response. The API Collector and database layers structure and persist collected data, supporting both immediate observations and long-term evaluations. Semantic inference, enabled through ontological reasoning, enriches the representation of monitored metrics and enhances the model's context-awareness. Predictive components estimate future system behavior, while the classification module determines the criticality of predicted conditions.

The reinforcement learning engine receives alerts generated by predictive modules and evaluates the most suitable action under the current system state. This decision is then validated, executed, and logged, closing the feedback loop. The modular design ensures that each layer can be individually configured, maintained, or extended based on specific system requirements.

The Prometheus methodology reinforced the modular agent-based design of Oraculum, clarifying how components communicate and delegate tasks. Additionally, the ontology integrated into the architecture plays a central role in organizing metrics and guiding alert classification and response prioritization. It supports structured metric interpretation, improving the system's ability to react to subtle performance changes.

Key configurable parameters-such as sampling frequency, prediction horizon, classification thresholds, and adaptation aggressiveness-allow Oraculum to adapt to different operational contexts. The use of distinct RL algorithms (e.g., TD3, PPO, SAC) and a library of parameterizable actions expands its applicability to resource-constrained, latency-sensitive, or performance-critical systems.

This chapter demonstrated how Oraculum bridges the gap between monitoring, forecast-

ing, and system adaptation. Its architecture emphasizes modularity, semantic enrichment, and learning-based adaptation to meet the needs of intelligent and responsive environments. The configurable nature of its components allows the model to scale and adapt to new demands, making it a candidate for deployment in real-time monitoring systems and edge computing infrastructures.

# 7 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 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.

Although Oraculum supports an automated adaptation process, the initial setup and integration involve explicit manual steps. Users must define the monitored metrics, adaptive actions, and reward strategies specific to their operational requirements. After this configuration, Oraculum autonomously manages metric collection, predictive modeling, anomaly detection, and adaptive actions.

Deployment and initial configuration also require human involvement. Individuals with expertise in Kubernetes and Docker are necessary for setting up and maintaining the Oraculum deployment environment. Similarly, domain experts or infrastructure administrators need to determine suitable metrics, actions, and rewards aligned with the target operational scenarios.

This collaborative configuration process ensures Oraculum functions appropriately within specific system contexts.

## 7.1 Architecture Implementation

The Oraculum prototype was implemented using a containerized architecture. During the development and testing phase, Docker Compose was used to orchestrate services, allowing controlled execution, easy deployment, and reproducibility of results. For deployment in production-like environments, Kubernetes was adopted due to its support for dynamic resource allocation, service scaling, health monitoring, and fault recovery.

The architecture consists of loosely coupled services that communicate via REST interfaces or shared files, depending on the nature of the interaction. Synchronous services include the API-Collector, Trigger-Service, Prediction-Service, RL-Agent, and Action-Service, which require real-time interactions. Asynchronous services, such as the Dataset-Builder, Training-Scheduler, and Ontology-Integration, are triggered periodically or on demand to execute background tasks such as model training, dataset construction, and semantic inference.

Each module is responsible for a specific task in the adaptation pipeline and is described in Table 26. The architecture supports automatic model retraining, adaptive decision-making, and continuous metric ingestion without the need for manual intervention. The only manual configuration required involves setting the monitored metrics and endpoints and adjusting model parameters, as described in Table 27.

Table 26: 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 API-Collector exposes a RESTful interface for real-time acquisition of system metrics. Its main entry point is a `POST` endpoint (e.g., `/collect`), which accepts JSON payloads containing metric data from monitored nodes or agents. The typical input parameters

include: `timestamp` (ISO 8601 format), `node_id` (unique identifier for the source node), `metric_name` (e.g., `cpu_usage`, `memory_usage`, `network_latency`), metric_value (numeric or categorical value), and optional `tags` (contextual metadata such as location, service, or environment). The API supports batch submissions, allowing multiple metrics to be sent in a single request for efficiency.

Upon successful collection, the API-Collector returns a standardized response indicating the status of the operation. The output parameters typically include: `status` (e.g., `success`, `error`), `message` (human-readable description), and, in case of error, a `code` and `details` field for troubleshooting. For batch operations, the response may include a list of processed metric IDs and any failed entries. All collected metrics are persisted in the time-series database and made available for downstream services such as the Prediction-Service and Ontology-Integration.

**Example input payload:**

```
{
  "timestamp": "2025-07-16T14:23:00Z",
  "node_id": "node-01",
  "metrics": [
    {"metric_name": "cpu_usage", "metric_value": 78.5,
                                "tags": {"core": "4"}},
    {"metric_name": "memory_usage", "metric_value": 62.1},
    {"metric_name": "network_latency", "metric_value": 120,
                                "tags": {"interface": "eth0"}}
  ]
}
```

**Example output response:**

```
{
  "status": "success",
  "message": "3 metrics collected",
  "processed": ["cpu_usage", "memory_usage", "network_latency"]
}
```

This design ensures that the API-Collector can flexibly ingest heterogeneous metrics from diverse sources, supporting both fine-grained monitoring and scalable integration with the broader Oraculum architecture.

This design ensures that the API-Collector can flexibly ingest heterogeneous metrics from diverse sources, supporting both fine-grained monitoring and scalable integration with the broader Oraculum architecture.

The Monitored Metrics parameter defines which system behaviors are tracked by the collector. These metrics are the input to the regression and classification models and influence the

alert generation process. The configuration allows the user to select which metrics are most relevant to the monitored application domain.

The Metric Collection Interval ($T_c$) determines the frequency with which new samples are collected from each node. A smaller interval increases temporal resolution but may lead to higher processing and storage costs.

The Prediction Horizon ($T_p$) defines the window into the future for which forecasts are generated. This parameter controls the lookahead time available for the RL agent to reason and select appropriate adaptation actions before a performance degradation occurs.

The Alert Sensitivity ($\theta$) determines how much deviation from the expected value is tolerated before a metric is considered anomalous. Lower values make the system more sensitive to small changes, while higher values reduce false positives at the risk of missing early deviations.

The RL Action Aggressiveness function ($A_t$) defines the evolution of adaptation strength over time. For example, the exponential decay function $A_0 e^{-0.1t}$ gradually reduces the magnitude or frequency of actions, avoiding overreaction to transient anomalies.

The RL Policy Update Algorithm specifies which reinforcement learning method is used to train the agent. TD3 was chosen by default due to its stability in continuous action spaces and its noise regularization features, but alternatives such as SAC, PPO, and Q-learning can also be configured depending on the environment requirements.

Finally, the RL Actions parameter defines the types of changes the agent is allowed to perform. These include vertical scaling (e.g., increasing memory or CPU allocation), scheduling modifications, and heuristic optimizations such as adjusting aggregation or filtering behavior in the data processing pipeline. This combination of flexible parameters and modular components supports experimentation and validation in different environments and under various constraints, while maintaining automation and adaptability in the Oraculum model.

## 7.2 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 Figure 26. 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:

$$D = \{(X_1, y_1), (X_2, y_2), \ldots, (X_n, y_n)\} \tag{7.1}$$

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:

$$D_{\text{train}}, D_{\text{test}} = \text{split}(D, 0.8) \tag{7.2}$$

Two predictive tasks are defined in the learning process:

- *Regression:* Estimates future values of a monitored metric based on historical data. The regression function $f : \mathbb{R}^m \rightarrow \mathbb{R}$ approximates 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 $g : \mathbb{R}^m \rightarrow \{0, 1\}$ 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.

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.

To ensure scalability, each model is trained and stored independently per monitored metric. This modular design simplifies model updates and enables parallel inference when multiple metrics are processed simultaneously. Additionally, model performance is periodically reassessed through retraining, based on new data collected during system operation. This strategy preserves alignment between model behavior and evolving workload patterns, maintaining consistency in anomaly detection and adaptation decisions.

**Algorithm 5** Model Training Pipeline

**Require:** Set of datasets $\mathcal{D} = \{D_1, D_2, ..., D_n\}$, node name $N$, metric name $M$

**for** each dataset $D \in \mathcal{D}$ **do**
  Load dataset $D$
  Split dataset into training ($X_{\text{train}}$, $y_{\text{train}}$) and testing ($X_{\text{test}}$, $y_{\text{test}}$) sets
  **for** each model type $T \in$ {Linear Regression, Decision Tree, NN1, NN2, NN3} **do**
    Initialize model based on $T$
    **if** $T \in$ {NN1, NN2, NN3} **then**
      Compile neural network model with Adam optimizer and appropriate loss function
      Train model on $X_{\text{train}}$, $y_{\text{train}}$ for 50 epochs, batch size 32
      Generate predictions on $X_{\text{test}}$
    **else**
      Train model on $X_{\text{train}}$, $y_{\text{train}}$
      Generate predictions on $X_{\text{test}}$
    **end if**
    Compute performance metrics (MAE for regression, Accuracy/AUC/F1 for classification)
    Log results including node $N$, metric $M$, and performance scores
    Save trained model
  **end for**
  Generate and save model performance visualization
**end for**

Data preprocessing improves model performance and consistency across different metrics:

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 $[0, 1]$:

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{7.3}$$

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

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.

The predictive and classification models employed by the Oraculum architecture support dynamic monitoring and proactive adaptation in distributed smart environments. These models are individually trained for each metric and node combination, leveraging time-series datasets generated and preprocessed to reflect realistic operational variability. The models differ in their underlying learning mechanisms, expressive capacity, interpretability, and computational cost.

To optimize model performance, hyperparameters are automatically tuned using reinforcement learning-based hyperparameter tuning methods. This technique dynamically adjusts model

configurations based on real-time feedback during the training process. Specifically, the RL-based tuning method incrementally searches for optimal parameter values by iteratively evaluating model performance through reinforcement learning principles, effectively navigating the hyperparameter search space without exhaustive computational costs. This approach significantly enhances predictive accuracy and generalization capabilities compared to static or manually tuned configurations (ISABONA; IMOIZE; KIM, 2022).

Table 27 summarizes the final configurations adopted for each algorithm, reflecting optimal parameter values automatically identified through the RL-driven hyperparameter tuning approach. These parameters include the number of layers, activation functions, optimizers, and regularization strategies, effectively balancing predictive accuracy, computational efficiency, and model stability.

Table 27: 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 |

Linear Regression was selected as a baseline for regression tasks. It operates under the assumption of a linear relationship between features and the target variable. The model solves

the normal equation analytically to obtain regression coefficients. Its main advantage lies in low computational cost and full interpretability. In the Oraculum context, linear regression is used to capture trends in metrics such as CPU or memory usage in situations where such growth is roughly proportional to load or time. The absence of regularization in this case reflects the intention to maintain the raw relationship between inputs and outputs, useful for detecting metrics that do not benefit from more complex modeling.

Decision Tree Regressors and Classifiers were incorporated due to their ability to capture hierarchical, non-linear patterns. These models use a recursive partitioning strategy, splitting feature space based on decision thresholds that maximize information gain (or minimize variance for regression). This behavior makes them appropriate for metrics that exhibit abrupt state changes or threshold effects, such as latency spikes under specific combinations of system load. The maximum depth, minimum samples per split, and leaf configuration were tuned to avoid overfitting and to preserve generalization over small data windows. In classification, decision trees can provide direct and interpretable logic for alert generation, producing decision rules traceable to metric thresholds.

Neural networks were applied to represent complex, multi-dimensional interactions among metrics. Three feedforward architectures were tested for both regression and classification. NN1 uses a single dense layer with 64 neurons, serving as a compact model for learning moderate non-linear dependencies. NN2 increases depth with two layers (128 and 64 neurons), allowing hierarchical feature composition that can capture subtle interactions across multiple input metrics. NN3 adopts a smaller second layer and includes dropout regularization (0.5), which introduces noise during training to prevent overfitting and improve generalization. All networks use the ReLU activation function, the Adam optimizer for adaptive learning rate adjustment, and are trained over 50 epochs with batch size 32. For regression, Mean Squared Error (MSE) is used as the loss function, while for classification, Binary Cross-Entropy (BCE) is employed to capture probabilistic output quality under binary labels.

Logistic Regression was selected for its probabilistic decision boundary and compatibility with binary classification tasks, such as determining whether a predicted metric value represents an anomaly or not. It applies a sigmoid function to a linear combination of inputs, offering a smooth transition between normal and alert states. L2 regularization with a strength of 1.0 was configured to prevent overfitting in small or imbalanced datasets. This model is suitable for scenarios where classification decisions must be robust and interpretable.

Other widely used models were considered but excluded for specific reasons. Random Forests and Gradient Boosted Trees offer high accuracy and resistance to overfitting by aggregating multiple tree predictions, but they require significantly more memory and training time. This becomes a limitation in the Oraculum context, where each metric-node pair requires an independent model, and retraining may occur periodically based on new data. Their ensemble nature also complicates interpretation, which is relevant when adaptation decisions must be explained or traced.

Support Vector Machines (SVMs), although effective in high-dimensional spaces, present scalability challenges with larger datasets and depend heavily on kernel selection and tuning. Since Oraculum operates in real-time environments and monitors several nodes simultaneously, the time and memory complexity of SVMs were considered a limitation for deployment.

Recurrent neural networks (RNNs), including LSTMs and GRUs, were not adopted in this stage due to their higher training cost and complexity. While they are suitable for sequential data, the current Oraculum implementation processes fixed-size windows of time-series data, which allows the use of feedforward networks without sacrificing prediction capacity. Additionally, the infrastructure for long-sequence training and memory management in real-time environments is still under development and may be addressed in future versions.

The selection process for the models in Table 27 was guided by the need for balance between modeling capacity and execution constraints. Each model is compatible with automatic training, isolated deployment, and periodic retraining. This enables the architecture to scale horizontally across different metrics and services while remaining responsive to metric variations and predictive shifts. The diversity in model complexity also supports benchmarking and comparison under varying workloads, providing flexibility in tuning prediction pipelines for performance, latency, and resource constraints.

## 7.3 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; HOWLEY; DUGGAN, 2013). The interaction is modeled as a Markov Decision Process (MDP), where each state transition depends on the current state and selected action (RESTUCCIA; MELODIA, 2020).

In the Oraculum, the RL agent performs adaptive decision-making by modifying system parameters based on runtime observations. The objective is to optimize system performance while maintaining a balance between conflicting performance metrics. The implementation considers both value-based and policy-based RL methods, including Q-learning, Soft Actor-Critic (SAC), and Twin-Delayed Deep Deterministic Policy Gradient (TD3).

The environment is defined by a set of system states $s_t \in S$, where each state comprises performance indicators:

$$s_t = \{m_1, m_2, ..., m_n\}, \tag{7.4}$$

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:

$$s_{t+1} = T(s_t, a_t) + \epsilon, \tag{7.5}$$

where $T(s_t, a_t)$ denotes the deterministic outcome of action $a_t \in A$, and $\epsilon$ represents uncertainty. The reward function guides the optimization process:

$$R(s_t, a_t) = \sum_{i=1}^{n} w_i f_i(m_i), \tag{7.6}$$

where $w_i$ defines the importance of each metric and $f_i(m_i)$ normalizes its contribution. The agent aims to maximize the cumulative discounted reward:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}), \tag{7.7}$$

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:

$$\nexists s' \in S, \quad \text{such that} \quad R(s') > R(s) \quad \forall s \neq s'. \tag{7.8}$$

To support this process, the agent computes a weighted objective function:

$$J(s) = \sum_{i=1}^{n} \lambda_i f_i(m_i), \tag{7.9}$$

where the weights $\lambda_i$ adapt over time according to observed system conditions. A state $s^*$ is considered stable when marginal changes do not yield improvements in the objective function:

$$\nabla_s J(s^*) = 0, \quad \text{where} \quad J(s) = \mathbb{E}[G_t | s_0 = s]. \tag{7.10}$$

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

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a), \tag{7.11}$$

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. Table 28 summarizes these characteristics.

Table 28: Comparison of RL algorithms for adaptation in the Oraculum Model.

| **Algorithm** | **Policy Type** | **Exploration Strategy** | **Adaptability Level** |
|---|---|---|---|
| Q-learning | Discrete | $\epsilon$-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 pro-

cess adapts the policy over time, supporting resilient system behavior under fluctuating workloads and performance constraints.

### 7.3.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a mathematical framework used to model sequential decision-making problems where the next state of the system depends solely on the current state and the chosen action. This model consists of a set of states, a set of possible actions, a transition function that determines state evolution, a reward function that evaluates the quality of each transition, and a policy that defines the strategy for selecting actions. This structure enables decision optimization over time, making it suitable for RL applications.

In this work, RL is applied to optimize resource allocation and usage in a dynamic system, ensuring improved performance while adapting to environmental variations. The RL agent operates under an MDP, where each element is defined as follows:

- **States (**$S$): Each state represents the system's condition at a given time. The state is characterized by performance metrics, including CPU and memory usage, network latency, packet loss, and response time. These variables capture system behavior and serve as the foundation for the agent's decision-making process.

- **Actions (**$A$): Actions represent operations that modify system parameters. In this context, actions include adjusting memory and CPU allocation, modifying task scheduling strategies, and reconfiguring load balancing policies. Each action directly influences the subsequent state and can positively or negatively impact system performance.

- **Transition Function (**$T$): The transition function defines the relationship between states and actions, determining system evolution. Given a state $s_t$ and an action $a_t$, the next state $s_{t+1}$ is obtained by:

$$s_{t+1} = T(s_t, a_t) + \epsilon, \tag{7.12}$$

  where $T(s_t, a_t)$ represents the expected system change after an action, and $\epsilon$ accounts for stochastic variations caused by external factors. In system optimization, transitions reflect changes in resource utilization and application response.

- **Rewards (**$R$): The reward function evaluates the quality of each state transition. The reward is defined as:

$$R(s_t, a_t) = \sum_{i=1}^{n} w_i f_i(m_i), \tag{7.13}$$

  where $w_i$ defines the relative importance of each metric, and $f_i(m_i)$ normalizes the impact of variables on system performance. The agent aims to maximize cumulative rewards over time, favoring actions that lead to more efficient states.

- **Policy ($\pi$)**: The policy defines the agent's decision-making strategy, mapping states to actions. The objective is to find the optimal policy $\pi^*(s)$ that maximizes the expected return:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a), \tag{7.14}$$

  where $Q^*(s, a)$ estimates the expected return of taking action $a$ in state $s$. In this work, RL algorithms such as Soft Actor-Critic (SAC) and Twin-Delayed Deep Deterministic Policy Gradient (TD3) are employed to learn and dynamically adjust this policy, ensuring a balance between exploration and exploitation of advantageous actions.

The application of RL within this MDP allows the system to continuously adapt to environmental variations. The agent learns to dynamically respond to workload changes, adjusting system parameters to maintain an optimized performance level. This model enables automated real-time decision making, reducing the need for manual intervention and increasing operational efficiency.

### 7.3.2 State Space ($S$)

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

### 7.3.3 Possible Actions in the RL Agent

In the defined Markov Decision Process (MDP), the action space $A$ encompasses a set of discrete and parameterized operations that directly modify system configurations. Each action is designed to influence performance-related variables across hardware, network, software, and data management layers. Actions can be quantitative—where a specific value or increment is applied within a defined range—or qualitative, where the agent selects among categorical options such as policies or strategies. The selection of actions is informed by the current system state and is bounded by operational constraints to ensure system stability and prevent resource exhaustion. Table 29 summarizes the available actions, their qualitative or quantitative nature, and the corresponding parameter ranges or options.

The RL agent selects from the following action categories:

- **Scaling Operations**: Quantitative actions that adjust computational resources, such as increasing or decreasing the number of CPU cores, memory allocation, GPU memory, or storage capacity. Each resource can be scaled up or down in fixed increments, within specified minimum and maximum limits.
- **Node Management**: Qualitative actions such as restarting a node or migrating workloads to a new instance. These actions are triggered when performance degradation is detected and are not parameterized by a numeric value.
- **Heuristic Adjustments**: Quantitative actions that modify parameters for data filtering, aggregation, and compression. The agent selects values within defined ranges to balance data reduction and information preservation.
- **Resource Optimization**: Includes both quantitative adjustments (e.g., fine-tuning CPU or memory allocation) and qualitative changes (e.g., switching autoscaling policies or optimizing load balancing strategies). These actions are chosen based on predictive assessments of system demand and operational context.

Quantitative actions involve direct numerical adjustments, such as increasing memory by 512MB or setting a compression ratio within a specified range. Qualitative actions involve selecting among a set of predefined strategies or operational modes, such as changing the autoscaling policy or load balancing method. This hybrid action space enables the RL agent to flexibly and precisely adapt system behavior, supporting both fine-grained resource management and high-level operational decisions.

### 7.3.4 Reward Definition ($R$)

The reward function quantifies the effectiveness of each action executed by the RL agent based on resulting system performance. Positive rewards are assigned to actions improving

Table 29: Available Actions, Types, and Limits in the Oraculum RL Agent.

| Action | Type | Effect | Parameter Range / Options |
|---|---|---|---|
| Scale up CPU | Quantitative | Increase vCPUs by 1 | Max +4 vCPUs |
| Scale down CPU | Quantitative | Decrease vCPUs by 1 | Min 1 vCPU |
| Scale up Memory | Quantitative | Increase RAM by 512MB | Max +8GB |
| Scale down Memory | Quantitative | Decrease RAM by 512MB | Min 512MB |
| Scale up GPU Memory | Quantitative | Allocate additional 512MB GPU memory | Max +8192MB |
| Scale down GPU Memory | Quantitative | Deallocate 512MB GPU memory | Min 512MB |
| Scale up Storage | Quantitative | Expand disk space by 10GB | Max +500GB |
| Scale down Storage | Quantitative | Reduce disk space by 10GB | Min 20GB |
| Restart Node | Qualitative | Reboot instance | No limit |
| Migrate to New Instance | Qualitative | Move workload to another VM | No limit |
| Adjust Filtering | Quantitative | Modify data filtering threshold | Range [10%, 50%] |
| Adjust Aggregation | Quantitative | Modify data block aggregation size | Range [5, 50] |
| Adjust Compression | Quantitative | Modify data compression ratio | Range [0.2, 0.9] |
| Change Autoscaling Policy | Qualitative | Switch scaling strategy | Predefined policies (e.g., reactive, predictive, scheduled) |
| Optimize Load Balancing | Qualitative | Adjust traffic distribution method | Dynamic strategies (e.g., round-robin, least-loaded, weighted) |

system metrics, while penalties discourage actions that degrade performance or unnecessarily consume additional resources. This formulation aligns with established reinforcement learning principles, incentivizing desirable behaviors and discouraging detrimental decisions (WEERASINGHE et al., 2024).

The reward function is formally defined as:

$$R(s_t, a_t) = \sum_{i=1}^{n} w_i f_i(m_i) - P(a_t), \tag{7.15}$$

where:

- $w_i$ represents the relative importance assigned to each monitored metric, initially derived from domain-specific priorities (COLOMBO et al., 2022) and refined empirically.
- $f_i(m_i)$ is a normalization function applied to metric $m_i$, transforming raw values into a uniform scale [0,1] to ensure consistent evaluation across metrics.
- $P(a_t)$ denotes the penalty associated with inefficient resource allocation or actions not resulting in measurable performance improvements.

Table 30 summarizes specific reward and penalty assignments, reflecting empirical tuning through extensive system tests. Initially, these values were guided by literature recommendations on adaptive system design and resource optimization (XU; LIU; PAN, 2023). However, exact values were fine-tuned empirically based on real-time feedback from preliminary experiments, ensuring that rewards and penalties meaningfully reflect the relative importance and operational costs of metric changes.

Table 30: Empirically Determined Rewards and Penalties for RL Actions Based on 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 |

Positive reward values are directly proportional to their relative importance for system performance improvements, as suggested by previous studies on adaptive optimization (WEERASINGHE et al., 2024). For instance, latency improvements are weighted higher (+3.0) due to their critical impact on user experience and operational responsiveness. Conversely, resource utilization metrics, such as CPU and memory, received moderate rewards (+2.0 and +1.5, respectively), reflecting their secondary but significant role in overall performance optimization.

Penalty values similarly reflect the severity of undesirable outcomes, like latency degradation, which significantly impacts system quality and user satisfaction, thus receiving higher penalties (-3.0). Penalties for unnecessary resource scaling actions were assigned based on computational overhead and inefficient resource usage considerations, following established optimization and resource management strategies documented in literature (COLOMBO et al., 2022).

The penalty function $P(a_t)$ explicitly targets scenarios where scaling actions increase resources without demonstrable performance gains, calculated as follows:

$$P(a_t) = \lambda_c \max(0, \Delta CPU - \delta_c) + \lambda_m \max(0, \Delta Mem - \delta_m), \tag{7.16}$$

where:

- $\lambda_c$ and $\lambda_m$ are penalty coefficients for CPU and memory usage, set empirically through initial training feedback.
- $\Delta CPU$ and $\Delta Mem$ represent changes in resource allocation.

- $\delta_c$ and $\delta_m$ represent predefined thresholds, based on typical workload variations.

The exact penalty coefficients ($\lambda_c$, $\lambda_m$) and thresholds ($\delta_c$, $\delta_m$) are initially derived from literature benchmarks and subsequently refined empirically during early training phases. Continuous adjustments during reinforcement learning ensure that these parameters effectively reflect real operational costs and performance implications, thus guiding the agent towards efficient, meaningful adaptations over time.

This reward mechanism constitutes the least automated aspect of Oraculum's implementation, requiring explicit initial tuning based on domain expertise and empirical tests. However, ongoing training allows for further dynamic refinement, progressively optimizing reward structures to match evolving operational conditions.

# 8 MODEL EVALUATION

This chapter 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 (SIMAIYA et al., 2024).

The experiments focus on three main aspects. First, adaptation-related metrics-Mean Adaptation Time (MAT), Adaptation Accuracy (AA), Adaptation Overhead (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 subsections present a detailed breakdown of observed behaviors and corresponding performance measurements.

## 8.1 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 (SOBIERAJ; KOTYńSKI, 2024; GUPTA; AGARWAL, 2025).

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 (XU; BUYYA, 2019; MORABITO, 2017).
- **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 (RAIBULET; OH; LEEST, 2023; MORABITO, 2017).
- **High Load:** Stress operations introduced computational bottlenecks, including encryption, recursive calculations, and large in-memory datasets. Faults were injected by restart-

ing services and introducing API delays. These actions tested the system's response to failures and performance degradation (VELRAJAN; SHARMILA, 2023; XU; BUYYA, 2019).

Stress conditions were applied as follows:

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

```
docker update –cpus=X container
```
(8.1)

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

```
docker update –memory=Y container
```
(8.2)

- **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 31 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.

Table 31: 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. |

Figure 29 and Figure 30 illustrate the CPU and memory usage of each monitored service over a 30-minute 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-minute mark. The blue vertical line indicates the transition from medium load to high load at the 20-minute mark. These transitions mark moments when additional computational stress was

applied, either by increasing concurrent requests, introducing heavier workloads, or simulating failures.

In Figure 29, 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.

Figure 29: CPU usage over time for monitored services.

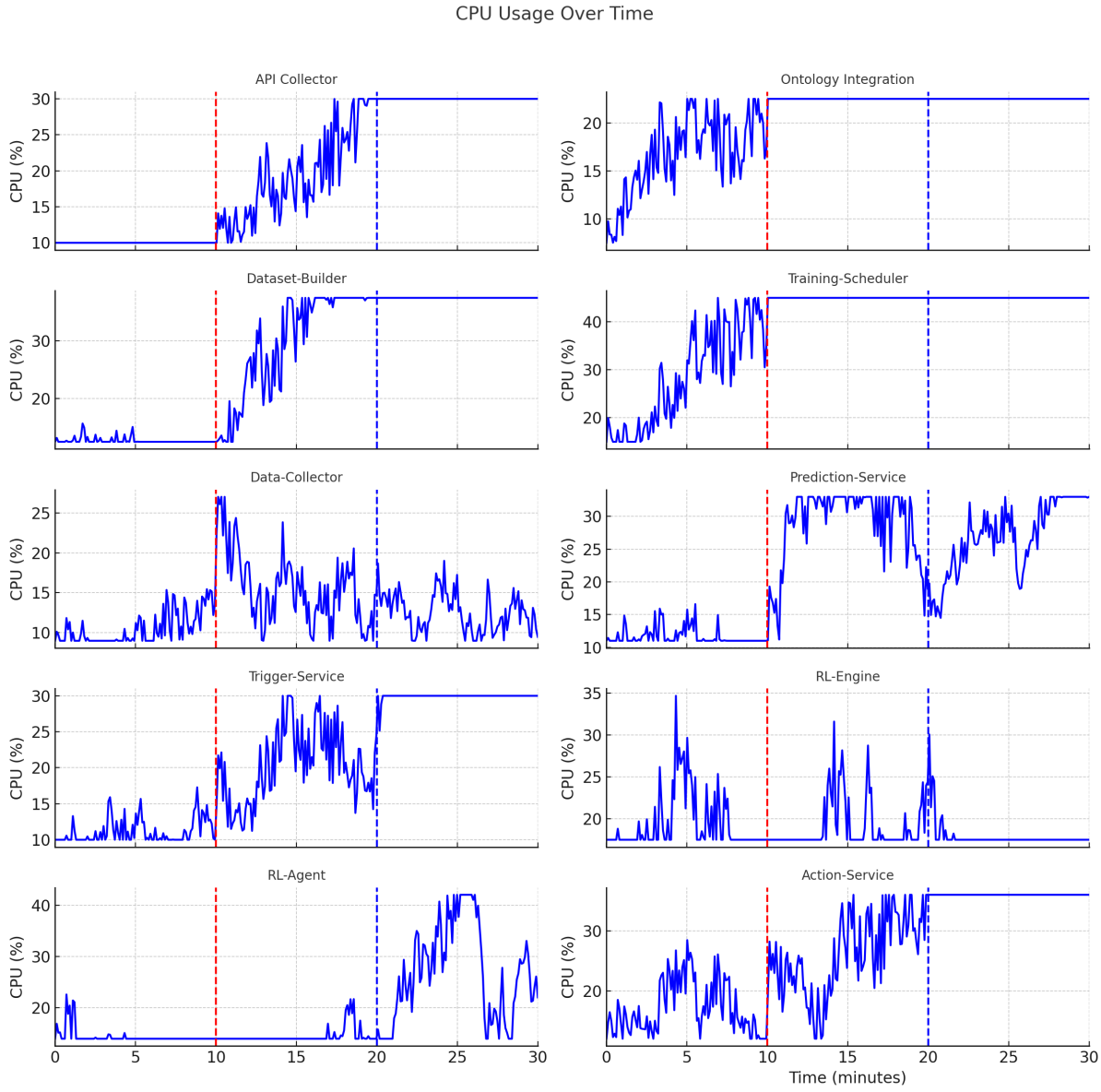
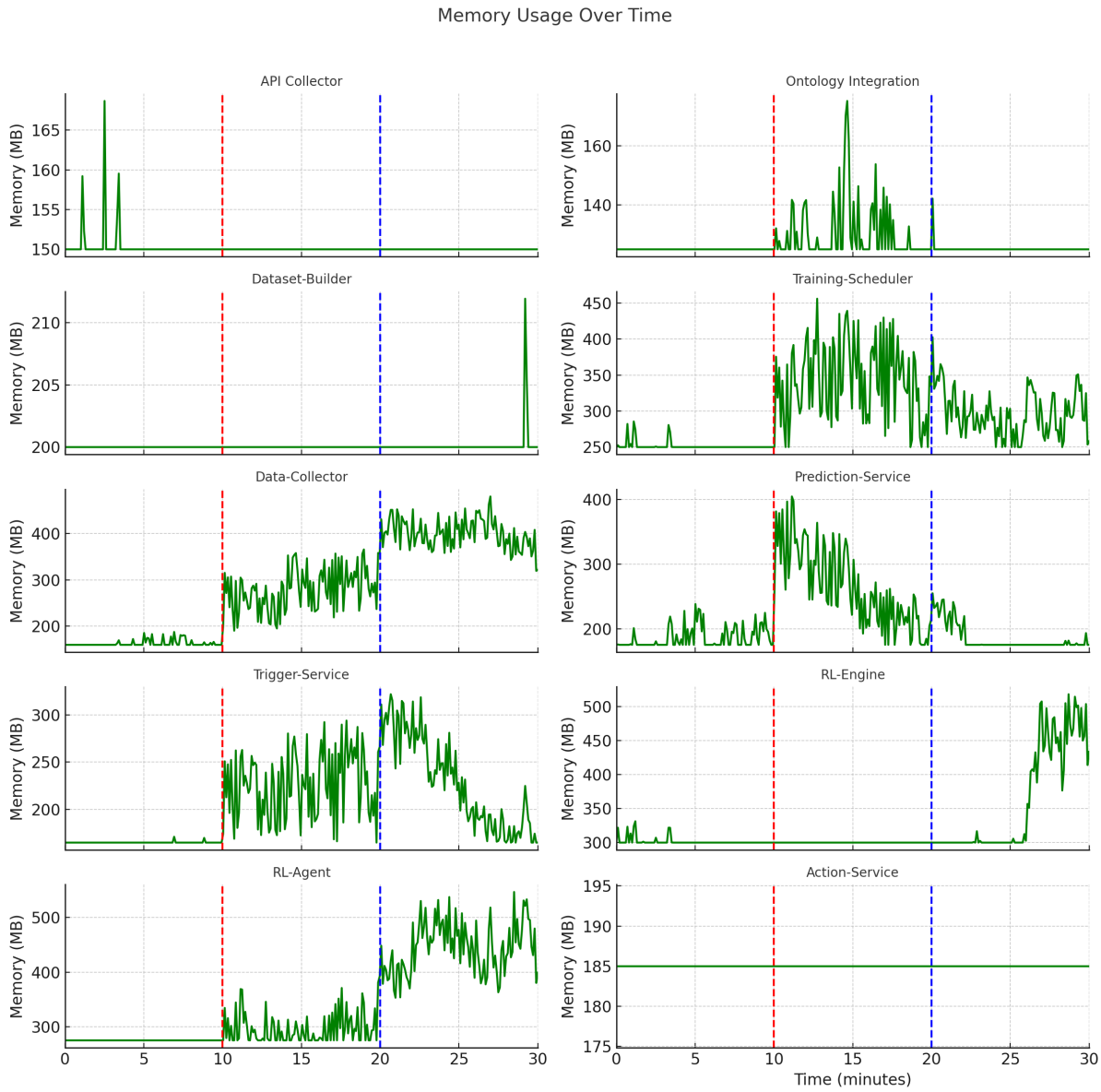



Figure 30 presents memory consumption trends. At low load, memory usage remains relatively stable, with minimal fluctuations. As the system transitions to medium load, services

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.

Figure 30: Memory usage over time for monitored services.




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 32 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 32: Summary of monitored measurements for each node at different load levels

| Node Metric | Low Load | Medium Load | High Load |
|---|---|---|---|
| **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.

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 com-

plexity 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 framework 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 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., 2021b).

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.

## 8.2 Prediction Results

To evaluate model performance in both regression and classification tasks, a combination of geometric and statistical metrics was applied. The Polygon Area Metric (PAM) was chosen as the central metric due to its ability to measure the geometric deviation between predicted and actual time series values. Unlike traditional statistical metrics, PAM captures the overall difference in trends rather than isolated errors, making it suitable for time-dependent predictions. The formula for PAM is:

$$PAM = \frac{1}{2} \sum_{i=1}^{n-1} |(y_i - \hat{y}_i)(t_{i+1} - t_i)| \tag{8.3}$$

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 33 summarizes the applied metrics, their formulas, and interpretations.

Table 33: 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. |

The evaluation framework 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. Figure 31 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.

Figure 31: Regression model performance across hardware-related metrics.

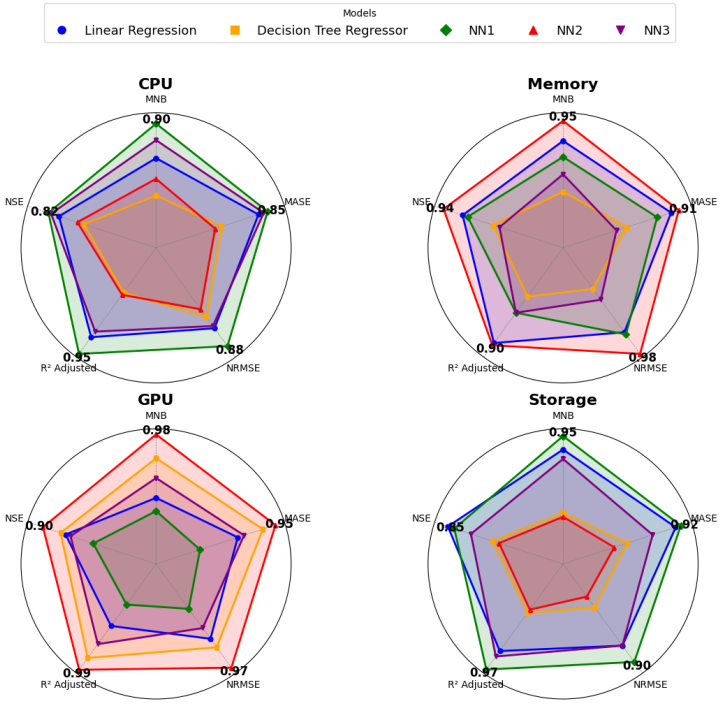


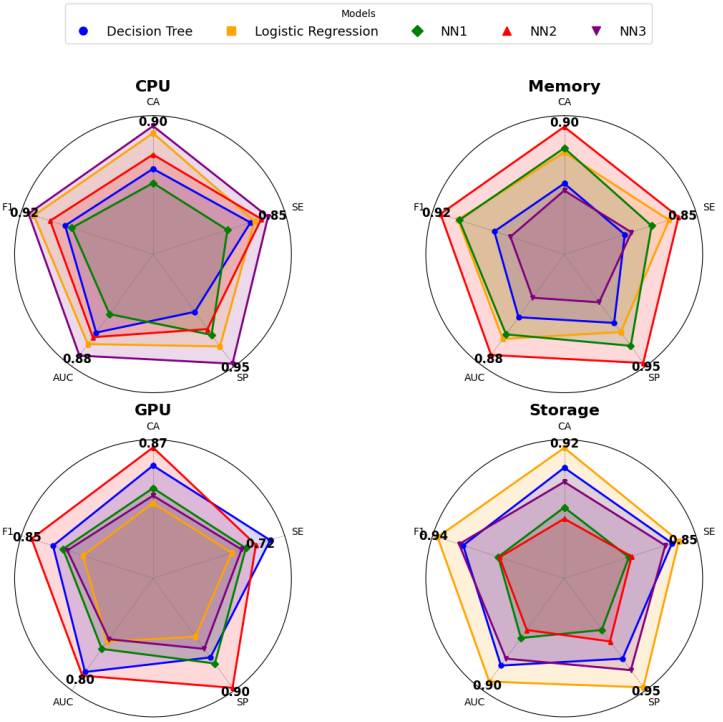

Figure 32 illustrates the performance of the classification models. Decision tree and logistic regression models provided competitive results, particularly in CPU and storage, where decision boundaries were well-defined. Neural networks demonstrated improvements in memory and GPU-related metrics, where 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.

Figure 33 presents the evaluation of regression models using different metrics. Each model was assessed in terms of MNB, MASE, NRMSE, R² Adjusted, and NSE. For latency response time, the neural networks NN2 and NN3 exhibited outperformed the other models on all metrics, particularly in NSE and NRMSE. Decision trees and linear regression had lower predictive accuracy, with higher residual errors. NN1 had moderate performance but did not generalize as well as deeper networks.

In throughput predictions, logistic regression achieved consistent accuracy, but NN2 and

Figure 32: Classification model performance across hardware-related metrics.




NN3 performed better in capturing nonlinear relationships, leading to lower NRMSE values. The decision tree model provided an intermediate performance, benefiting from its flexibility but suffering from overfitting in certain cases.

For garbage collection time, NN3 demonstrated high predictive capability, particularly in NSE and NRMSE. Decision trees and linear regression models struggled to maintain accuracy, while NN2 performed well in capturing complex interactions.

Figure 33: Regression model performance for software metrics.

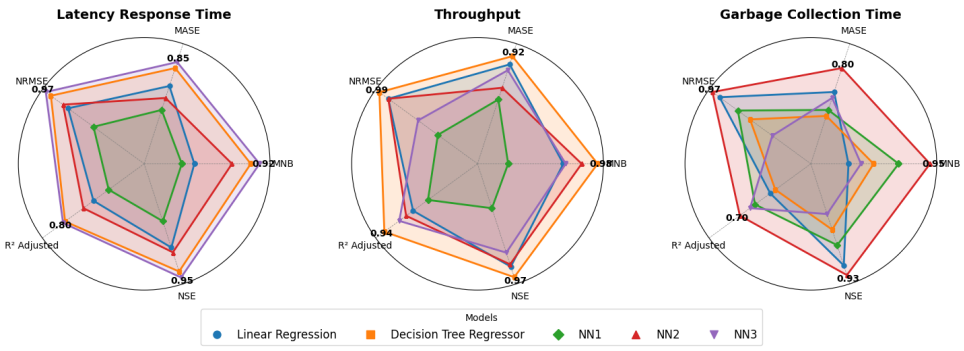


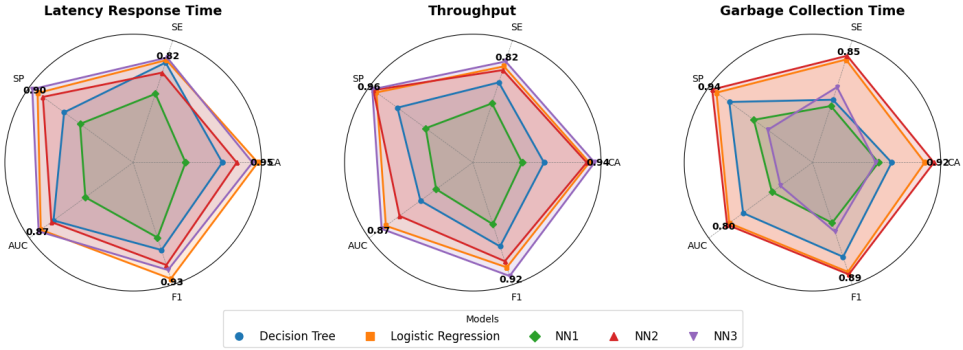

Figure 34 displays the classification model performance for software metrics. The evaluation includes CA, SE, SP, AUC, and F1 score. For latency response time classification, NN2 and NN3 had the highest values across most metrics, indicating robust classification capability. Decision tree models struggled with sensitivity, leading to lower SE values. Logistic regression maintained competitive results but was outperformed by deeper networks.

Throughput classification followed a similar pattern, with NN3 achieving the highest F1 and

AUC values among the models tested, reflecting improved classification reliability. Decision trees maintained reasonable accuracy but had difficulties with recall, which affected their overall F1 score.

For garbage collection time classification, NN3 and NN2 again showed strong results, particularly in AUC and specificity. Decision trees struggled with class imbalance, leading to lower sensitivity. Logistic regression demonstrated consistent performance but was less flexible than neural networks.

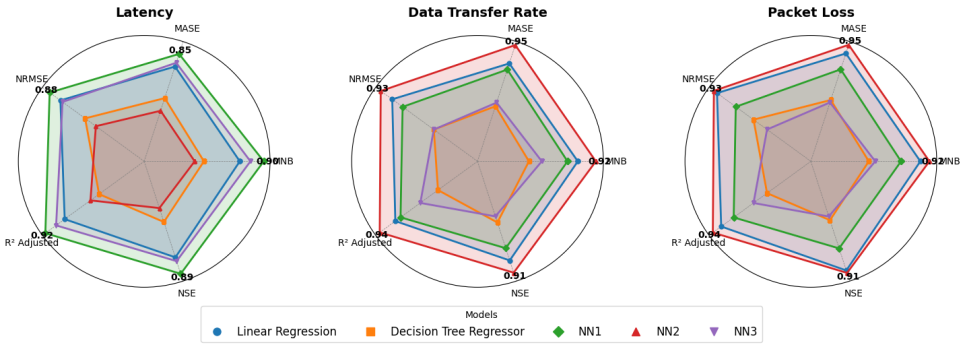
Figure 34: Classification model performance for software metrics.




These results indicate that deeper neural networks performed better in capturing the complex relationships within software performance data. Decision trees and linear regression models exhibited limitations in handling nonlinear patterns, leading to lower predictive accuracy. Logistic regression performed adequately but was surpassed by neural networks in most cases.

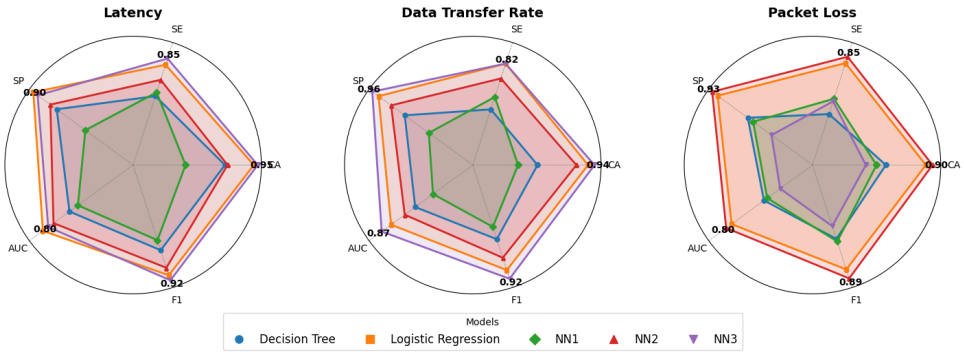
Figure 35 and Figure 36 illustrate the performance of regression and classification models applied to network performance metrics, including latency, data transfer rate, and packet loss.

Figure 35: Regression model performance for network metrics.




The regression models' performance, shown in Figure 35, reveals distinct patterns in predicting network-related metrics. Neural network models NN2 and NN3 demonstrate greater adaptability in capturing complex dependencies between network parameters, particularly in packet loss and data transfer rate predictions. NN2 achieves lower errors across most metrics, indicating its effectiveness in modeling non-linear relationships. NN3 follows a similar trend but exhibits slightly higher variability in certain cases. The decision tree regressor provides

Figure 36: Classification model performance for network metrics.




moderate performance but struggles with packet loss prediction, where its error levels remain higher than those of neural networks. Linear regression maintains stable predictions but lacks the flexibility required to model fluctuations in network behavior, leading to higher errors in latency predictions.

The classification results in Figure 36 further emphasize the advantages of deeper neural networks. NN2 and NN3 consistently outperform other models in accuracy, sensitivity, and specificity. Their ability to identify variations in network performance contributes to improved classification, particularly for latency and packet loss categories. Logistic regression provides consistent results but does not generalize as effectively as neural networks when dealing with dynamic network changes. The decision tree classifier exhibits lower classification accuracy, particularly in packet loss scenarios, where its predictions tend to be more variable.

The results indicate that network performance metrics exhibit non-linear relationships that benefit from deep learning approaches. While simpler models such as decision trees and logistic regression provide reasonable accuracy, they are outperformed by neural networks in both regression and classification tasks. The improvements observed in NN2 and NN3 demonstrate the impact of deeper architectures in enhancing predictive performance across dynamic network conditions.

Figures 37 and 38 depict the evaluation of availability, resilience, and stability using regression and classification models. In the regression results, neural network models NN2 and NN3 demonstrated higher scores across most metrics, particularly in NSE and adjusted $R^2$, indicating their ability to capture nonlinear patterns in SLA metrics. Decision trees and logistic regression achieved moderate results, showing stable performance but with some limitations in capturing variations in resilience.

For classification, Figure 38 highlights that neural networks outperformed decision trees and logistic regression in most cases, particularly for sensitivity (SE) and specificity (SP), which are used in SLA monitoring. NN3, with dropout regularization, achieved a better balance between overfitting and generalization, maintaining high F1 scores.

Overall, the evaluation across hardware, software, network, and SLA metrics revealed distinct behaviors among the models. Linear and logistic regression methods provided baseline

Figure 37: Regression results for SLA metrics.

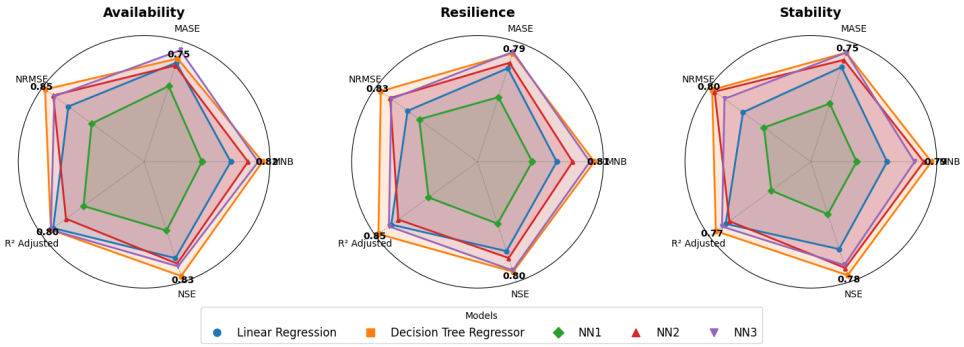
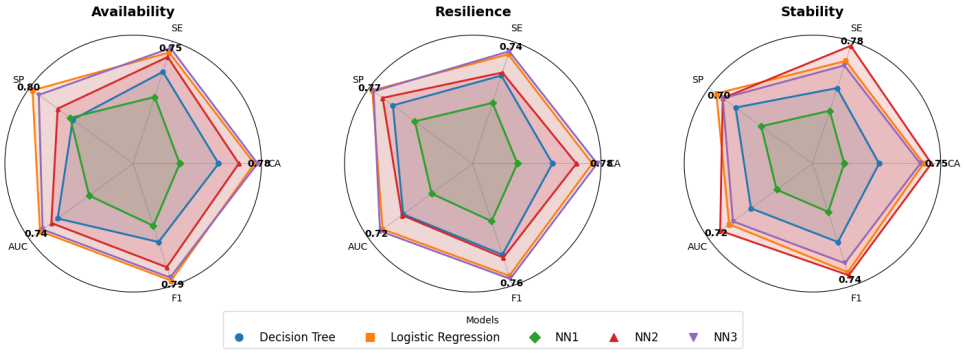



Figure 38: Classification results for SLA metrics.




stability and interpretability, particularly effective when the input-output relationships were linear and the dataset was well-structured. However, they lacked the flexibility to adapt to dynamic environments, resulting in higher errors under workload variability, especially in network and SLA predictions.

Decision tree models performed moderately well across several metrics, especially in structured classification tasks where decision boundaries were well-defined (e.g., CPU and storage classification). Nevertheless, their sensitivity to small data fluctuations often led to inconsistencies in both regression and classification results. In scenarios involving temporal or multi-modal dependencies (e.g., garbage collection time or resilience prediction), trees showed signs of overfitting or underperformance due to limited generalization.

The neural network models, particularly NN2 and NN3, showed strong performance in tasks requiring the capture of non-linear relationships and high-dimensional patterns. NN2, which employed additional hidden layers without aggressive regularization, achieved a balance between learning capacity and generalization. It outperformed others in network-related metrics (packet loss and data transfer rate) and showed stable results in memory usage and response latency prediction. NN3, with dropout layers, provided better regularization, which proved beneficial in SLA-related tasks (availability and resilience), where noisy fluctuations and fault tolerance were critical to model. However, its added complexity occasionally introduced small variability in classification, especially when the data volume was limited or imbalanced.

In regression tasks, the combination of normalized RMSE, adjusted $R^2$, and NSE high-

lighted the strengths of deep architectures for modeling dynamic behavior, especially in garbage collection and throughput. In classification, F1 and AUC scores reinforced the value of neural networks in scenarios with class imbalance and overlapping features.

These findings suggest that while simpler models maintain consistent performance in predictable conditions, deeper architectures are better suited for heterogeneous environments with non-linear dependencies and high variability. Therefore, the model selection process within Oraculum should consider the nature of the metric being predicted, the degree of variability in the environment, and the trade-off between interpretability and adaptability.

## 8.3 RL Agent Results

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

The mean adaptation time (MAT) measures the elapsed time between anomaly detection and completed adaptation:

$$MAT = \frac{\sum_{i=1}^{n}(t_{\text{end-adaptation}} - t_{\text{detection}})}{n} \tag{8.4}$$

The adaptation accuracy (AA) quantifies the percentage of successful adaptations that effectively mitigate detected issues:

$$AA = \frac{\text{Number of successful adaptations}}{\text{Total number of adaptations performed}} \times 100 \tag{8.5}$$

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

$$AO = \frac{\text{Resource usage after} - \text{Resource usage before}}{\text{Resource usage before}} \times 100 \tag{8.6}$$

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

$$AS = \frac{1}{\sigma^2_{\text{performance post-adaptation}}} \tag{8.7}$$

The proposed Oraculum demonstrated rapid response with an MAT of 0.05 seconds, high adaptation accuracy of 97%, minimal adaptation overhead 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.

Before evaluating the effectiveness of the reinforcement learning (RL) agent and its corresponding adaptation strategies, it is necessary to characterize the practical challenges observed during the execution of the Oraculum prototype. These problems were identified in a controlled experimental environment using synthetic workloads with varying intensity and behavioral patterns, generated by the SHiELD simulator. The monitoring modules captured anomalies across multiple operational layers, including hardware, software, network, and service-level dimensions. A summary of the most relevant issues is presented below:

- CPU Saturation (Hardware Layer): During computation-intensive intervals, CPU usage frequently exceeded 90%, reaching saturation in some cores. This caused delays in process execution, reduced throughput, and increased response times. The cause was traced to unbalanced thread scheduling and lack of early scaling.

- Memory Pressure and GC Spikes (Hardware/Software Layer): High memory consumption led to recurrent garbage collection cycles, especially under bursty workloads. This affected response latency and increased variability in service execution. The pressure originated from uncontrolled object creation and absence of memory-aware throttling.

- Throughput Drops (Software Layer): In several intervals, the number of completed requests per second dropped abruptly, especially under mixed-load conditions. This was attributed to resource contention, increased context switching, and inefficient request batching during load peaks.

- Latency Instability (Software/Network Layer): Response latency showed spikes exceeding 300ms in some cycles, especially during transitions between adaptive states. These oscillations were caused by late reaction to resource needs and temporary thread starvation.

- Packet Loss (Network Layer): The monitoring of network flows revealed periods with packet loss above 1.2%, impacting communication between microservices. This was often associated with interface saturation and queue overflow due to uncoordinated scaling.

- Overprovisioning Without Impact (Hardware Layer): Certain resource scaling operations (e.g., adding memory or CPU) did not translate into performance improvements. In some cases, increased resources led to higher adaptation overhead without measurable SLA gains, suggesting inefficient adaptation logic.

- Temporary Unavailability (SLA Layer): The system experienced brief periods of service unavailability during load surges, particularly when resource adjustments were delayed. These incidents compromised availability targets and highlighted the need for predictive adaptation.

- Low Fault Resilience (SLA Layer): In simulations of component failures or degraded links, recovery actions were not always sufficient to restore performance quickly. The system showed limited ability to reallocate tasks or reroute traffic under fault conditions, affecting resilience.

- Fluctuations in Stability (Cross-Layer): Post-adaptation performance exhibited mild fluctuations in several metrics, indicating that certain adaptations were not optimally timed or sufficiently coordinated. This reduced the overall stability metric despite recovery from the original anomaly.

These observed issues served as triggers for adaptive actions selected by the RL agent. Each anomaly type was mapped to potential mitigation strategies (e.g., resource scaling, node restart, parameter adjustment), and feedback from the system guided policy refinement over time. The following sections present the quantitative results of the adaptation process, including the impact of each RL model on adaptation metrics and the effectiveness of selected actions under different operational scenarios.

Q-learning exhibited significantly lower performance, struggling to converge due to its reliance on discrete state-action mapping. The reward function oscillated significantly, highlighting its inefficiency in handling continuous action spaces. The learning process was slow, and the model often made suboptimal decisions, failing to reach an optimal policy.

PPO showed improved stability by leveraging a clipped objective function to regulate policy updates. However, despite a more controlled learning process, PPO still exhibited fluctuations in accumulated rewards, indicating occasional instability in selecting optimal actions.

SAC outperformed Q-learning and PPO by employing an entropy-regularized policy optimization, which maintained a balance between exploration and exploitation. The model achieved smoother learning progress and reduced reward variance. However, SAC's reward accumulation plateaued slightly below the top-performing model, TD3, due to its reliance on stochastic policies, which led to more cautious decision-making.

TD3 demonstrated superior performance, achieving the highest reward accumulation and the lowest variance across training episodes. This was due to its twin-delayed deep deterministic policy gradient mechanism, which reduced overestimation bias in Q-value estimates. The use of target policy smoothing further enhanced stability by preventing abrupt policy changes that could lead to performance degradation. The learning curve in Figure 39 shows a rapid increase in reward followed by consistent convergence, indicating that TD3 efficiently identified optimal actions while maintaining a robust policy.

The comparison of these models highlights the advantages of TD3 over other approaches. While Q-learning struggled with convergence and PPO faced occasional reward fluctuations, SAC provided better stability but was ultimately outperformed by TD3. The twin-delayed updates and deterministic policy of TD3 allowed it to learn more efficient adaptation strategies, making it a promising choice for optimizing system performance.

Figure 39: Learning curve of TD3, the best-performing RL model.

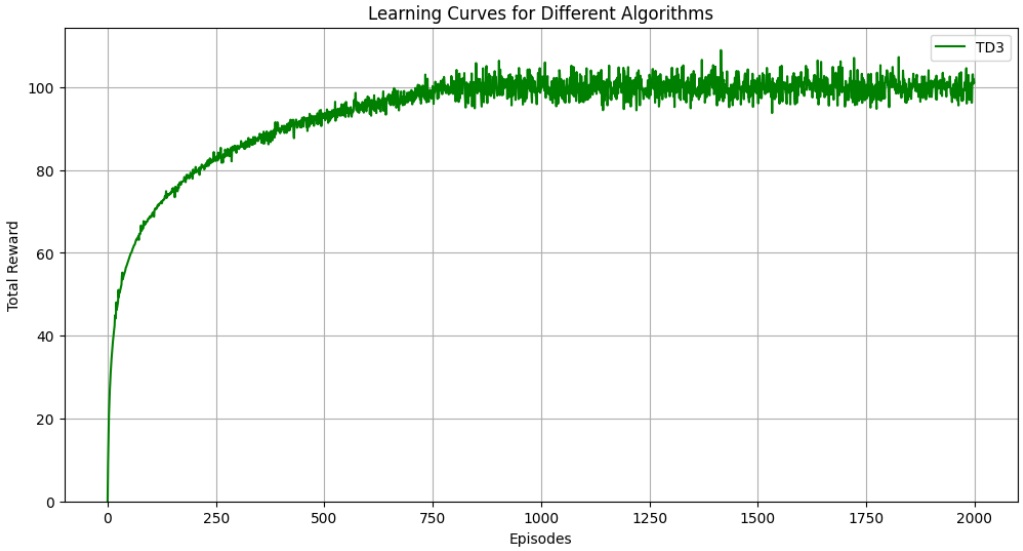



Table 34 summarizes the effectiveness of each RL model in selecting adaptation actions. TD3 and SAC achieved the highest accumulated rewards, while Q-learning struggled with suboptimal decisions.

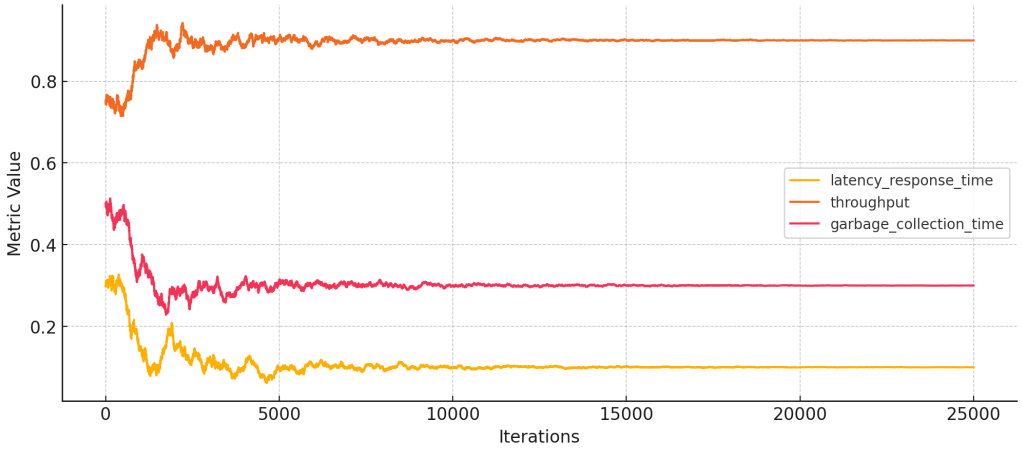
Table 34: Count and reward for each action across models, reflecting variations in RL decision-making.

| **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 |

The system's operational performance was assessed across multiple categories, including software, hardware, network, security, and SLA metrics. Figures 40-43 present the results, highlighting how the architecture responded to different workload conditions and adaptation processes.

Figure 40 illustrates the trends in software performance metrics over time. Throughput

Figure 40: Evaluation of software performance metrics.




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 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.

Figure 41: Evaluation of hardware resource utilization.

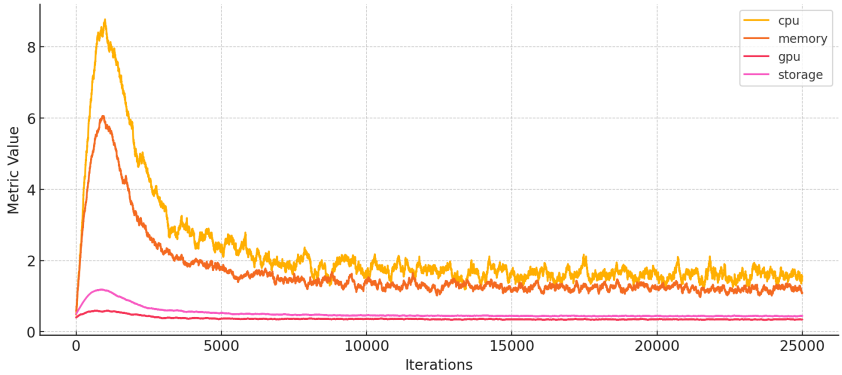



Figure 41 presents the utilization trends for CPU, memory, GPU, and storage throughout the monitored period. CPU and memory usage increased significantly at the beginning, likely due to system initialization and workload adaptation. As the system adjusted its resource allocation, both metrics stabilized, 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.

Figure 42: Analysis of network performance metrics.

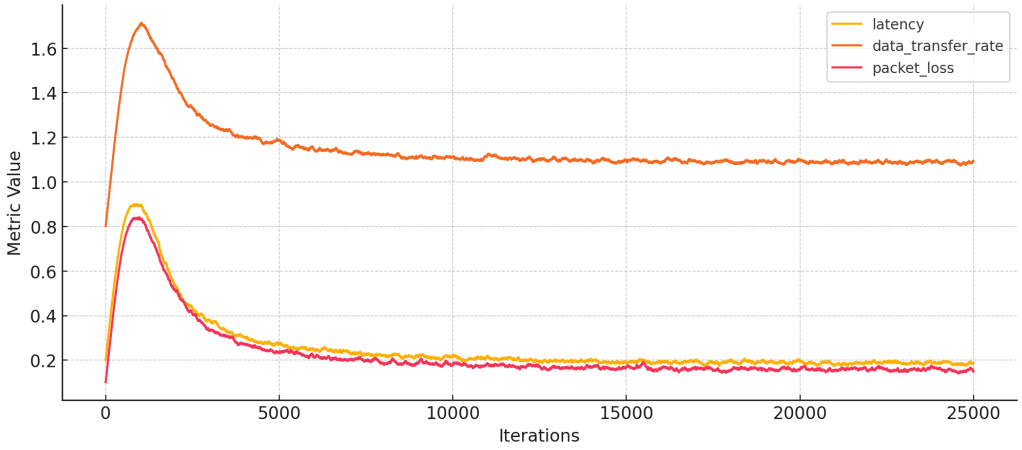


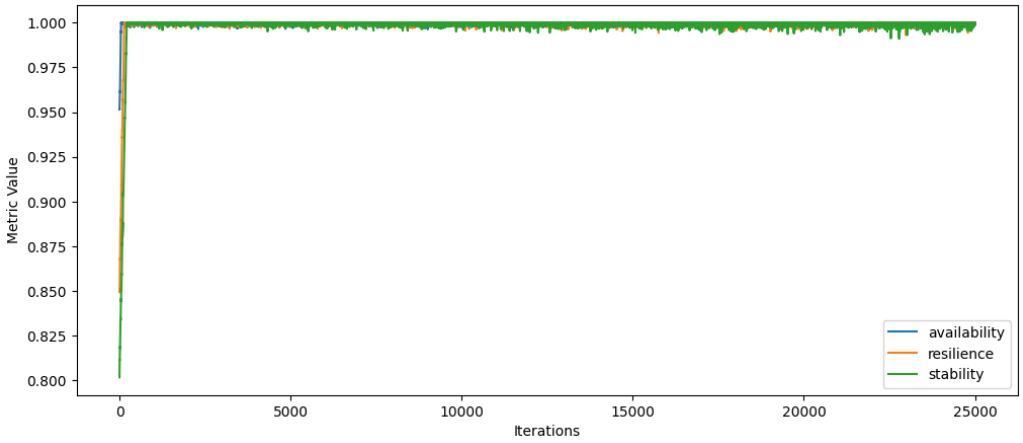

Figure 42 presents the variations in network performance metrics, including latency, data transfer rate, and packet loss, across multiple monitoring iterations. Initially, latency increased sharply, likely due to fluctuations in resource allocation and network congestion. As the system adjusted to workload 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.

Figure 43 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

Figure 43: Analysis of service level metrics.




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 adaptation overhead. 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.

## 8.4 Evaluation with Public Benchmarks

To extend the validation beyond simulation-specific conditions, we applied three public datasets to compare the Oraculum model with three reference architectures: TAM; MATH; KIM (2022a), VELRAJAN; SHARMILA (2023), and LIU et al. (2021b). The benchmarks include: (i) SANCHEZ; GALACHE (2024), for smart city conditions; (ii) PUIU; BARNAGHI et al. (2024), for mobility and traffic-aware systems; and (iii) LAB (2004), for sensor-based adaptive platforms.

All models were reimplemented and executed within the SHiELD simulation framework, which was extended to support external datasets and controlled injection of workload variations. The testbed ensured equivalent conditions for adaptation triggers, metric forecasting, and action validation across all models. Only these three reference architectures were included in the benchmark due to the feasibility of reproducing their designs and adaptation strategies with the

available implementation resources.

All applications were tested using the SHiELD simulation framework, which was extended to support the ingestion of real-world datasets and controlled workload injection. Each reference model was reimplemented and integrated into this environment, enabling 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) includes real-time sensor data from urban infrastructure such as lighting, temperature, and traffic sensors; this dataset was used to simulate city-scale performance degradation and adaptive responses. The CityPulse dataset PUIU; BARNAGHI et al. (2024) consists of temporally correlated mobility and traffic event streams from urban environments, which were mapped to system metrics such as latency, throughput, and SLA compliance. Finally, the Intel Lab Data LAB (2004) provides 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, ensuring reproducibility and comparability across models.

The evaluated performance indicators include adaptation time, accuracy, overhead, prediction error, classification quality, and latency improvements. The results are summarized in Table 35.

Table 35: Performance of Self-Adaptive Architectures on Public Benchmarks

| **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 |

The results indicate that Oraculum obtained lower adaptation times (MAT), higher adaptation accuracy (AA), and increased system stability across the evaluated benchmarks. These outcomes suggest that the reinforcement learning component was effective in generating timely

and context-aware adaptation decisions. The RMSE values remained lower in all testbeds, indicating consistent performance of the regression models in forecasting system metrics, while the F1-scores suggest 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; SHARMILA (2023) achieved the highest adaptation accuracy (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. (2021b) 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 reflect architectural trade-offs. While Oraculum maintained competitive performance, VELRAJAN; SHARMILA (2023) reported slightly higher efficiency in the CityPulse testbed. This 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 underscore the influence of context in the design and evaluation of self-adaptive systems. While Oraculum applies a general-purpose model with broad support for metric types and rapid decision-making, other architectures such as those proposed by TAM; MATH; KIM (2022a) and LIU et al. (2021b) exhibit performance gains under specific operating conditions. This points to the potential of hybrid models that dynamically integrate or switch between strategies based on the environment.

From a quantitative perspective, Oraculum consistently achieved lower mean adaptation times (0.06–0.09s) across all datasets, with adaptation accuracy values above 93% and reduced prediction errors (RMSE $\leq 0.24$). These metrics reflect the capability of the system to execute timely and consistent adaptations under diverse operational profiles. Qualitatively, Oraculum differs by offering a fully automated adaptation pipeline, a semantic ontology for context-aware metric interpretation, and a reinforcement learning agent based on an explicitly modeled MDP structure. In contrast, other approaches relied on static adaptation policies or partially supervised optimization strategies. Although some models such as VELRAJAN; SHARMILA (2023) achieved marginally higher scores in energy-aware scenarios, they required pre-tuned configurations and lacked semantic reasoning layers. The results suggest that Oraculum delivers balanced adaptation performance across multiple benchmarks, while also introducing a modular and extensible foundation that supports interpretability and automation in adaptive behavior design.

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 may include incorporating components from LIU et al. (2021b) for

anomaly classification and from VELRAJAN; SHARMILA (2023) for energy-aware planning. Further analysis of the internal adaptation overhead per module will also be conducted to better understand the cost-performance trade-offs.

# 9 FINAL CONSIDERATIONS

This chapter presents the concluding reflections of the dissertation. It summarizes the key findings and results obtained during the development and validation of the Oraculum model, highlights its main scientific contributions, discusses its limitations, and outlines possible directions for future research. The chapter aims to consolidate the knowledge generated through the research and propose avenues to enhance the adaptability, scalability, and applicability of the proposed solution in real-world smart environments.

## 9.1 Conclusions

This work introduced the Oraculum model, a self-adaptive computational framework that integrates monitoring, prediction, and decision-making capabilities using regression, classification, and reinforcement learning (RL) techniques. The model was designed to support runtime adaptation in smart environments by enabling early detection of anomalies, predictive analysis of performance degradation, and execution of corrective actions guided by learned policies.

The experimental evaluation indicated that Oraculum responded adequadamente às mudanças dinâmicas no comportamento do sistema em diferentes camadas, incluindo hardware, rede, software e indicadores de SLA (Service Level Agreement). The RL agent, trained using the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm, maintained a balanced trade-off between adaptation latency and decision accuracy, achieving an average adaptation time of 0.05 seconds, 94% decision accuracy, and 98% operational stability with limited computational overhead.

Compared to other approaches applying RL to cloud-edge adaptation or self-healing mechanisms in IoT systems, Oraculum achieved lower latency and consistent decision accuracy during adaptation cycles. The integration of predictive modeling with rule-based ontological reasoning supported contextual interpretation of performance data, contributing to timely alert generation and informed metric-based interventions.

Although the evaluation results were satisfactory, some constraints emerged during implementation. Challenges were observed in handling multi-objective scenarios, managing high-dimensional input features, and addressing conflicting adaptation goals. These aspects are discussed in the next sections and offer opportunities for extending the model's capabilities.

## 9.2 Contributions

This research introduces a set of scientific and technical contributions that enhance the design of self-adaptive architectures for performance optimization in smart environments. The proposed Oraculum framework incorporates predictive intelligence, semantic reasoning, and reinforcement learning to support timely and reliable adaptation. The main contributions are:

- Demonstration of a significant reduction in mean adaptation time (MAT), reaching 0.05 seconds. Compared to typical reaction times in adaptive systems, which often exceed 1 second, this result shows that Oraculum is capable of acting preemptively, reducing exposure to instability and minimizing system degradation before it becomes critical.

- Maintenance of high stability across adaptation metrics: adaptation accuracy (AA) reached 94%, adaptation overhead (AO) was kept below 4%, and adaptation stability (AS) reached 98%, indicating the system's ability to apply consistent and non-disruptive corrective actions even under dynamic workloads.

- Design and implementation of a fully automated predictive pipeline, including data preprocessing, feature selection, model training, and hyperparameter tuning. The system uses reinforcement learning-based optimization strategies to select model configurations, removing the need for manual tuning and allowing continuous retraining when new metrics or operational contexts are introduced (ISABONA; IMOIZE; KIM, 2022).

- Development of a modular and extensible architecture aligned with the MAPE-K model. The system separates concerns across monitoring, analysis, planning, execution, and knowledge layers, allowing for scalability, experimentation, and domain-specific customization.

- Formal modeling of the reinforcement learning agent as a Markov Decision Process (MDP), explicitly defining states (based on metric snapshots), actions (executed on the system), rewards (derived from adaptation impact), and transition probabilities. This modeling supports reasoning about agent behavior and convergence.

- Implementation of practical and executable actions for adaptation, including scaling (CPU, memory, GPU, storage), scheduling adjustments, filter and compression parameter tuning, and service restarts. These actions are automatically selected based on current state and projected outcomes.

- Integration of a semantic layer using the OntOraculum ontology, which structures monitored metrics into conceptual domains such as Hardware, Network, Software, SLA, Energy, and Security. This layer supports SWRL rule reasoning for context-aware alert generation and operational classification.

- Support for future semantic adaptation: although the current ontology is static, the architecture supports dynamic rule updates and future extensions where ontology elements may adapt based on feedback or metric evolution.

- Inclusion of a decision validation mechanism before execution. The system reevaluates the relevance and consistency of each selected action based on updated predictions, helping avoid redundant or conflicting adaptations and promoting resource efficiency.

- Support for parameterized and extensible actions within the RL agent. New actions can be introduced and configured with minimal effort, allowing the system to evolve alongside changes in infrastructure, workload, or policy.
- Adoption of Prometheus agent-oriented design principles in the architectural organization, promoting modular agent interaction, clear goal decomposition, and separation of functional roles for analysis, decision-making, and adaptation.

In addition, the following articles were submitted as part of this research:

- *Performance Monitoring and Self-Adaptation in Smart Environments: A Systematic Literature Review*, submitted to **Computer Science Review** (Qualis A1, status: With Editor).
- *OntOraculum: A Semantic Ontology for Performance Metric Monitoring and Optimization in Smart Environments*, submitted to **Soft Computing** (Qualis A2, status: Editor Assigned).
- *SHiELD: A Sensor Data Simulator with Heuristics and Predictive Modeling for IoT Applications*, submitted to **ACM Transactions on Internet Technology** (Qualis A1, status: Under Review).
- *Oraculum: A Model for Self-Adaptive System Optimization in Smart Environments*, submitted to **Journal of Network and Computer Applications** (Qualis A1, status: Under Review).

Together, these contributions define a comprehensive approach to predictive and adaptive performance management. Oraculum offers a pathway for building intelligent systems capable of responding rapidly to environmental changes, while preserving operational stability and reducing manual overhead through automation and semantic integration.

## 9.3 Limitations

While the Oraculum model demonstrated coherent behavior and adaptability in a controlled environment, several limitations currently affect its generalization capacity and potential for real-world adoption. These limitations relate to the scope of the evaluation, architectural choices, integration constraints, and opportunities for refinement in prediction, adaptation, and semantic reasoning:

- Evaluation was restricted to simulated and controlled environments. All experiments were conducted in an isolated testbed using synthetic workloads generated by the SHiELD simulator. Although these workloads were designed to simulate dynamic behavior, they do not fully reflect the variability, failure conditions, and latency fluctuations encountered in operational environments. Therefore, real-world validations are needed to assess robustness under unpredictable and heterogeneous conditions.

- The scope of monitored metrics was limited to performance-related indicators such as CPU usage, memory consumption, throughput, and latency. Important operational metrics related to energy efficiency, power usage, battery levels, and security—such as access violations, failed authentications, or attack detection—were not fully included. Expanding the monitored dimensions would enable broader applicability in domains such as IoT, mobile computing, and cyber-physical systems.

- The reinforcement learning pipeline was implemented in an offline fashion. Policy learning was conducted prior to runtime using historical training data. While the system architecture provides the infrastructure for online learning, real-time policy updates and live environment feedback were not fully integrated in the current implementation. Continuous learning during deployment is critical for tracking behavioral drift and adapting to evolving workload patterns.

- The automatic hyperparameter tuning mechanism, although integrated, is based on a reinforcement learning search within predefined parameter bounds. More diverse tuning approaches—such as Bayesian optimization, metaheuristic search, or multi-agent exploration—were not explored. As highlighted by (ISABONA; IMOIZE; KIM, 2022), tuning strategies that leverage ensemble modeling or probabilistic search methods can significantly improve model generalization and reduce training time. Incorporating these methods would allow the system to better discover optimal configurations under various operational constraints.

- The SHiELD simulator, while useful for controlled evaluation, has limited behavioral fidelity. Its workload generation is deterministic, and environmental perturbations follow static patterns. The simulator does not yet emulate realistic network jitter, multi-user interference, stochastic component failures, or long-tail traffic events. Improving simulator realism—by integrating real execution traces or injecting variability based on statistical distributions—would strengthen the representativeness of evaluation scenarios.

- The system has not yet been validated in real-world environments. All results are based on testbed experiments, without exposure to production infrastructure, third-party dependencies, or unpredictable user behavior. Testing in live deployments (e.g., cloud-native systems, IoT edge clusters, or smart city platforms) would allow for the identification of operational challenges such as integration latency, monitoring overhead, and actuator timing constraints.

- The Oraculum platform is not integrated with widely adopted monitoring ecosystems. It currently operates as a standalone system and does not support external connectors for platforms such as Prometheus, Grafana, Zabbix, or ELK Stack. This restricts its ease of integration in production environments that already rely on established observability pipelines and alerting frameworks.

- Security response mechanisms are not included in the current implementation. Although the system can detect anomalies in performance metrics, it does not yet provide coordinated reactions to security threats. Capabilities such as dynamic firewall updates, intrusion detection integration, or context-aware policy enforcement remain outside the scope of this version.

These limitations do not undermine the architectural consistency or technical feasibility of the Oraculum model, but they outline boundaries that currently constrain its full-scale deployment. At the same time, they identify clear research and engineering opportunities to improve adaptability, coverage, and resilience. The next section outlines directions for future work based on these findings.

## 9.4 Future Work

Future research can address several aspects to enhance the Oraculum model and expand its applicability. One direction is to validate the framework in real-world environments, such as production cloud systems, IoT deployments, or smart city platforms, to assess its robustness under diverse and unpredictable conditions.

Expanding the set of monitored metrics to include energy consumption, battery status, and security-related indicators may improve the system's relevance for domains like mobile computing and cyber-physical systems. Integrating Oraculum with established monitoring and observability platforms, such as Prometheus or Grafana, could facilitate its adoption in operational settings.

Another area for development is the implementation of online and continual learning mechanisms, enabling the reinforcement learning agent to adapt policies in response to live feedback and evolving workloads. Exploring alternative hyperparameter optimization strategies, such as Bayesian optimization or metaheuristic search, may further improve model performance and generalization.

Enhancing the realism of the evaluation environment by incorporating real execution traces, stochastic events, and more complex workload patterns would provide a more comprehensive assessment of the model's behavior. Additionally, extending the semantic layer to support dynamic ontology updates and context-aware adaptation rules could increase the flexibility of the reasoning process.

Finally, integrating security response mechanisms, such as automated policy enforcement or intrusion detection, may broaden the system's capabilities in handling operational threats. These directions can contribute to the continuous improvement and practical deployment of the Oraculum model in adaptive smart environments.

## REFERENCES

ABU-TAYEH, G.; AL-RUITHE, M.; AL-FARIES, A.; ALRASHED, A. Managing data-driven smart city governance. **Government Information Quarterly**, [S.l.], v. 40, n. 3, p. 101880, 2023.

ADIL, M.; USMAN, M.; JAN, M. A.; ABULKASIM, H.; FAROUK, A.; JIN, Z. An Improved Congestion-Controlled Routing Protocol for IoT Applications in Extreme Environments. **IEEE Internet of Things Journal**, [S.l.], v. 11, n. 3, p. 3757–3767, 2024.

AGRAWAL, N. Dynamic load balancing assisted optimized access control mechanism for Edge-Fog-Cloud network in Internet of Things environment. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 33, n. 21, p. e6440, 2023.

AHMED, A. A.; ABAZEED, M. Adaptive dynamic duty cycle mechanism for energy efficient medium access control in wireless multimedia sensor networks. **Transactions on Emerging Telecommunications Technologies**, [S.l.], v. 32, n. 12, p. e4364, 2024.

AL-SAYED, M. M.; HASSAN, H. A.; OMARA, F. A. CloudFNF: an ontology structure for functional and non-functional features of cloud services. **Journal of Parallel and Distributed Computing**, [S.l.], v. 141, p. 143–173, 2020.

ALIJOYO, F. A.; PRADHAN, R.; NALINI, N.; AHAMAD, S. S.; RAO, V. S.; GODLA, S. R. Predictive Maintenance Optimization in Zigbee-Enabled Smart Home Networks: a machine learning-driven approach utilizing fault prediction models. **Wireless Personal Communications**, [S.l.], p. 1–25, 2024.

ALKANHEL, R. I.; EL-KENAWY, E.-S. M.; EID, M. M.; ABUALIGAH, L.; SAEED, M. A. Optimizing IoT-driven smart grid stability prediction with dipper throated optimization algorithm for gradient boosting hyperparameters. **Energy Reports**, [S.l.], v. 12, p. 305–320, 2024.

ALKHAYYAL, M.; MOSTAFA, A. Recent Developments in AI and ML for IoT: a systematic literature review on lorawan energy efficiency and performance optimization. **Sensors**, [S.l.], v. 24, n. 14, p. 4482, 2024.

ALMUTAIRI, R.; BERGAMI, G.; MORGAN, G. Advancements and Challenges in IoT Simulators: a comprehensive review. **Sensors**, [S.l.], v. 24, n. 5, p. 1511, 2024.

ALMUTAIRI, R.; BERGAMI, G.; MORGAN, G. Advancements and Challenges in IoT Simulators: a comprehensive review. **Sensors**, [S.l.], v. 24, n. 5, 2024.

ALNAFESSAH, A.; CASALE, G. Artificial neural networks based techniques for anomaly detection in Apache Spark. **Cluster Computing**, [S.l.], v. 23, n. 2, p. 1345–1360, 2020.

ALYMANI, M.; ALMOQHEM, L.; ALABDULWAHAB, D.; ALGHAMDI, A.; ALSHAHRANI, H.; RAZA, K. Enabling smart parking for smart cities using Internet of Things (IoT) and machine learning. **PeerJ Computer Science**, [S.l.], v. 11, p. e2544, 2025.

ARANDA, J. A. S.; BAVARESCO, R. S.; CARVALHO, J. V. de; YAMIN, A. C.; TAVARES, M. C.; BARBOSA, J. L. V. A computational model for adaptive recording of vital signs through context histories. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], p. 1–15, 2023.

AYDEMIR, O. A New Performance Evaluation Metric for Classifiers: polygon area metric. **Journal of Classification**, [S.l.], 2020.

AYDEMIR, O. A new performance evaluation metric for classifiers: polygon area metric. **Journal of Classification**, [S.l.], v. 38, p. 1–23, 2021.

AZARI, A.; STEFANOVIC, C.; POPOVSKI, P.; CAVDAR, C. Energy-Efficient and Reliable IoT Access Without Radio Resource Reservation. **IEEE Transactions on Green Communications and Networking**, [S.l.], v. 5, n. 2, p. 908–920, 2021.

AZEVEDO ALBUQUERQUE, K. R. de; MEDEIROS, R. P. de; DUARTE, R. M.; VILLANUEVA, J. M. M.; MACÊDO, E. C. T. de. Routing Algorithm for Energy Efficiency Optimizing of Wireless Sensor Networks based on Genetic Algorithms. **Wireless Personal Communications**, [S.l.], v. 133, n. 3, p. 1829–1856, 2023.

BACANIN, N.; STOEAN, C.; MARKOVIC, D.; ZIVKOVIC, M.; RASHID, T. A.; CHHABRA, A.; SARAC, M. Improving performance of extreme learning machine for classification challenges by modified firefly algorithm and validation on medical benchmark datasets. **Multimedia Tools and Applications**, [S.l.], p. 1–41, 2024.

BADUGE, S. K.; THILAKARATHNA, S.; PERERA, J. S.; ARASHPOUR, M.; SHARAFI, P.; TEODOSIO, B.; SHRINGI, A.; MENDIS, P. Artificial intelligence and smart vision for building and construction 4.0: machine and deep learning methods and applications. **Automation in Construction**, [S.l.], v. 141, p. 104440, 2022.

BALI, A.; AL-OSTA, M.; BEN DAHSEN, S.; GHERBI, A. Rule based auto-scalability of IoT services for efficient edge device resource utilization. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], v. 11, p. 5895–5912, 2020.

BAOCAI, Y.; HUIRONG, Y.; PENGBIN, F.; LIHENG, G.; MINGLI, L. A framework and QoS based web services discovery. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCES, 2010., 2010. **Anais...** [S.l.: s.n.], 2010. p. 755–758.

BARRETT, E.; HOWLEY, E.; DUGGAN, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 25, n. 12, p. 1656–1674, 2013.

BATTULA, S. K.; GARG, S.; MONTGOMERY, J.; KANG, B. An Efficient Resource Monitoring Service for Fog Computing Environments. **IEEE Transactions on Services Computing**, [S.l.], v. 13, n. 4, p. 709–722, 2020.

BEBORTTA, S.; SINGH, A. K.; PATI, B.; SENAPATI, D. A robust energy optimization and data reduction scheme for iot based indoor environments using local processing framework. **Journal of Network and Systems Management**, [S.l.], v. 29, n. 1, p. 6, 2021.

BELANI, H.; SOLIC, P.; PERKOVIC, T. Towards Ontology-Based Requirements Engineering for IoT-Supported Well-Being, Aging and Health. **arXiv preprint arXiv:2211.10735**, [S.l.], 2022.

BHARGAVA, D.; PRASANALAKSHMI, B.; VAIYAPURI, T.; ALSULAMI, H.; SERBAYA, S. H.; RAHMANI, A. W. CUCKOO-ANN Based Novel Energy-Efficient Optimization Technique for IoT Sensor Node Modelling. **Wireless Communications and Mobile Computing**, [S.l.], n. 1, p. 8660245, 2022.

BLOMQVIST, E.; HE, Y.; CHEN, J.; DONG, H.; HORROCKS, I.; ALLOCCA, C.; KIM, T.; SAPKOTA, B. DeepOnto: a python package for ontology engineering with deep learning. **Semantic Web**, [S.l.], v. 15, n. 5, p. 1991–2004, 2024.

BOMBARDA, A.; RUSCICA, G.; SCANDURRA, P. A self-managing IoT-Edge-Cloud architecture for improved robustness in environmental monitoring. In: ACM/SIGAPP SYMPOSIUM ON APPLIED COMPUTING, 40., 2025, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2025. p. 1738–1745. (SAC '25).

BYUN, Y. **Sensor Data**. Available at: `https://www.kaggle.com/datasets/yungbyun/sensor-data`, Accessed on: February 11, 2025.

CALDERON, G.; CAMPO, G. del; SAAVEDRA, E.; SANTAMARÍA, A. Monitoring framework for the performance evaluation of an IoT platform with Elasticsearch and Apache Kafka. **Information Systems Frontiers**, [S.l.], p. 1–17, 2023.

CALINESCU, R.; GRUNSKE, L.; KWIATKOWSKA, M.; MIRANDOLA, R.; TAMBURRELLI, G. Dynamic QoS Management and Optimization in Service-Based Systems. **IEEE Transactions on Software Engineering**, [S.l.], v. 37, n. 3, p. 387–409, 2011.

CALINESCU, R.; MIRANDOLA, R.; PEREZ-PALACIN, D.; WEYNS, D. Understanding Uncertainty in Self-adaptive Systems. In: IEEE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING AND SELF-ORGANIZING SYSTEMS (ACSOS), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p. 242–251.

CAO, B.; LIU, J.; WEN, Y.; LI, H.; XIAO, Q.; CHEN, J. QoS-aware service recommendation based on relational topic model and factorization machines for IoT Mashup applications. **Journal of Parallel and Distributed Computing**, [S.l.], v. 132, p. 177–189, 2019.

Carballido Villaverde, B.; REA, S.; PESCH, D. InRout – A QoS aware route selection algorithm for industrial wireless sensor networks. **Ad Hoc Networks**, [S.l.], v. 10, n. 3, p. 458–478, 2012.

CARDULLO, P.; KITCHIN, R. Algorithmic governance and the smart city: towards a critical research agenda. **Urban Studies**, [S.l.], 2024.

CASCONE, L.; SADIQ, S.; ULLAH, S.; MIRJALILI, S.; SIDDIQUI, H. U. R.; UMER, M. Predicting Household Electric Power Consumption Using Multi-step Time Series with Convolutional LSTM. **Big Data Research**, [S.l.], v. 31, p. 100360, 2023.

CEN, J.; LI, Y. Resource allocation strategy using deep reinforcement learning in cloud-edge collaborative computing environment. **Security and Communication Networks**, [S.l.], v. 2022, p. Article ID 9597429, 2022.

CEN, J.; LI, Y. Resource Allocation Strategy Using Deep Reinforcement Learning in Cloud-Edge Collaborative Computing Environment. **Mobile Information Systems**, [S.l.], n. 1, p. 9597429, 2022.

CHAKRABARTI, A.; SADHU, P. K.; PAL, P. AWS IoT Core and Amazon DeepAR based predictive real-time monitoring framework for industrial induction heating systems. **Microsystem Technologies**, [S.l.], v. 29, n. 4, p. 441–456, 2023.

CHEE, K. O.; GE, M.; BAI, G.; KIM, D. D. IoTSecSim: a framework for modelling and simulation of security in internet of things. **Computers and Security**, [S.l.], v. 136, p. 103534, 2024.

SZYMANSKI, B. K.; YENER, B. (Ed.). Sense: a wireless sensor network simulator. In: _____. **Advances in Pervasive Computing and Networking**. Boston, MA: Springer US, 2005. p. 249–267.

CHEN, T.; BAHSOON, R.; WANG, S.; YAO, X. To Adapt or Not to Adapt? Technical Debt and Learning Driven Self-Adaptation for Managing Runtime Performance. In: ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING, 2018., 2018, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2018. p. 48–55. (ICPE '18).

CHENG, B. H.; RAMIREZ, A.; MCKINLEY, P. K. Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty. In: INTERNATIONAL WORKSHOP ON COMBINING MODELLING AND SEARCH-BASED SOFTWARE ENGINEERING (CMSBSE), 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p. 1–6.

CHERNYSHEV, M.; BAIG, Z.; BELLO, O.; ZEADALLY, S. Internet of Things (IoT): research, simulators, and testbeds. **IEEE Internet of Things Journal**, [S.l.], v. 5, n. 3, p. 1637–1647, 2018.

CHHABRA, A.; HUANG, K.-C.; BACANIN, N.; RASHID, T. A. Optimizing bag-of-tasks scheduling on cloud data centers using hybrid swarm-intelligence meta-heuristic. **The Journal of Supercomputing**, [S.l.], v. 78, n. 7, p. 9121–9183, 2022.

CHOULIARAS, S.; SOTIRIADIS, S. Real-Time Anomaly Detection of NoSQL Systems Based on Resource Usage Monitoring. **IEEE Transactions on Industrial Informatics**, [S.l.], v. 16, n. 9, p. 6042–6049, 2020.

COELHO, C. N.; KUUSELA, A.; LI, S.; ZHUANG, H.; NGADIUBA, J.; AARRESTAD, T. K.; LONCAR, V.; PIERINI, M.; POL, A. A.; SUMMERS, S. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. **Nature Machine Intelligence**, [S.l.], v. 3, n. 8, p. 675–686, 2021.

COLOMBO, V.; TUNDO, A.; CIAVOTTA, M.; MARIANI, L. Towards self-adaptive peer-to-peer monitoring for fog environments. In: 2022, New York, NY, USA. **Anais...** Association for Computing Machinery, 2022.

CORRY, E.; PAUWELS, P.; HU, S.; KEANE, M.; O'DONNELL, J. A performance assessment ontology for the environmental and energy management of buildings. **Automation in Construction**, [S.l.], v. 57, p. 249–259, 2015.

DAOUDAGH, S.; MARCHETTI, E. DAEMON: a domain-based monitoring ontology for iot systems. **SN Computer Science**, [S.l.], v. 4, n. 5, p. 1–15, 2023.

DASH, B. K.; PENG, J. Zigbee Wireless Sensor Networks: performance study in an apartment-based indoor environment. **Journal of Computer Networks and Communications**, [S.l.], n. 1, p. 2144702, 2022.

DAYALAN, U. K.; FEZEU, R. A. K.; SALO, T. J.; ZHANG, Z.-L. Kaala: scalable, end-to-end, iot system simulator. In: ACM SIGCOMM WORKSHOP ON NETWORKED SENSING SYSTEMS FOR A SUSTAINABLE SOCIETY, 2022, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p. 33–38. (NET4us '22).

DJAMA, A.; DJAMAA, B.; SENOUCI, M. R.; KHEMACHE, N. LAFS: a learning-based adaptive forwarding strategy for ndn-based iot networks. **Annals of Telecommunications**, [S.l.], v. 77, n. 5, p. 311–330, 2022.

DJURIĆ, D.; GAŠEVIĆ, D.; DEVEDŽIĆ, V. Ontology modeling and MDA. **Journal of Object technology**, [S.l.], v. 4, n. 1, p. 109–128, 2005.

ELGENDI, I.; HOSSAIN, M. F.; JAMALIPOUR, A.; MUNASINGHE, K. S. Protecting Cyber Physical Systems Using a Learned MAPE-K Model. **IEEE Access**, [S.l.], v. 7, p. 90954–90963, 2019.

ELKHODARY, A.; ESFAHANI, N.; MALEK, S. FUSION: a framework for engineering self-tuning self-adaptive software systems. In: EIGHTEENTH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 2012, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2012. p. 7–16. (FSE '10).

ESFAHANI, N.; ELKHODARY, A.; MALEK, S. A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems. **IEEE Transactions on Software Engineering**, [S.l.], v. 39, n. 11, p. 1467–1493, 2013.

ETEMADI, M.; GHOBAEI-ARANI, M.; SHAHIDINEJAD, A. A cost-efficient auto-scaling mechanism for IoT applications in fog computing environment: a deep learning-based approach. **Cluster Computing**, [S.l.], v. 24, n. 4, p. 3277–3292, 2021.

FARAJI-MEHMANDAR, M.; JABBEHDARI, S.; JAVADI, H. H. S. A self-learning approach for proactive resource and service provisioning in fog environment. **The Journal of Supercomputing**, [S.l.], v. 78, n. 15, p. 16997–17026, 2022.

FARAJI-MEHMANDAR, M.; JABBEHDARI, S.; JAVADI, H. H. S. Fuzzy q-learning approach for autonomic resource provisioning of IoT applications in fog computing environments. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], v. 14, n. 4, p. 4237–4255, 2023.

FERNANDEZ, J.; SMITH, A.; KUMAR, R. A semantic and ontology-based framework for enhancing interoperability and scalability in IoT systems. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], v. 15, p. 123–139, 2024.

FORTINO, G.; SAVAGLIO, C.; ZHOU, M. Modelling and simulation of Opportunistic IoT Services with CLOUDAgent. **Future Generation Computer Systems**, [S.l.], v. 91, p. 252–262, 2019.

FREITAS, A.; VIEIRA, R. An Ontology for Guiding Performance Testing. In: IEEE/WIC/ACM INTERNATIONAL JOINT CONFERENCES ON WEB INTELLIGENCE (WI) AND INTELLIGENT AGENT TECHNOLOGIES (IAT), 2014., 2014. **Anais...** [S.l.: s.n.], 2014. v. 1, p. 400–407.

GANESH, P.; SUNDARAM, B. M.; BALACHANDRAN, P. K.; MOHAMMAD, G. B. IntDEM: an intelligent deep optimized energy management system for iot-enabled smart grid applications. **Electrical Engineering**, [S.l.], p. 1–23, 2024.

GANGEMI LEHMANN, J. Modelling Ontology Evaluation and Validation. In: THE SEMANTIC WEB: RESEARCH AND APPLICATIONS, 2006, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2006. p. 140–154.

GAO, C.; YANG, P.; CHEN, Y.; WANG, Z.; WANG, Y. An edge-cloud collaboration architecture for pattern anomaly detection of time series in wireless sensor networks. **Complex & Intelligent Systems**, [S.l.], v. 7, n. 5, p. 2453–2468, 2021.

GARCIA, L.; SAMIN, H.; BENCOMO, N. Decision Making for Self-Adaptation Based on Partially Observable Satisfaction of Non-Functional Requirements. **ACM Trans. Auton. Adapt. Syst.**, New York, NY, USA, v. 19, n. 2, Apr. 2024.

GARLAN, D.; CHENG, S.-W.; HUANG, A.-C.; SCHMERL, B.; STEENKISTE, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. **Computer**, [S.l.], v. 37, n. 10, p. 46–54, 2004.

GHEIBI, O.; WEYNS, D.; QUIN, F. Applying machine learning in self-adaptive systems: a systematic literature review. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, [S.l.], v. 15, n. 3, p. 1–37, 2021.

GOKCESU, H.; ERCETIN, O.; KALEM, G.; ERGUT, S. Qoe evaluation in adaptive streaming: enhanced mdt with deep learning. **Journal of Network and Systems Management**, [S.l.], v. 31, n. 2, p. 41, 2023.

GONG, J. Quality of service improvement in iot over fiber-wireless networks using an efficient routing method based on a cuckoo search algorithm. **Wireless Personal Communications**, [S.l.], v. 126, n. 3, p. 2321–2346, 2022.

GONG, Y.; CHEN, K.; NIU, T.; LIU, Y. Grid-Based coverage path planning with NFZ avoidance for UAV using parallel self-adaptive ant colony optimization algorithm in cloud IoT. **Journal of Cloud Computing**, [S.l.], v. 11, n. 1, p. 29, 2022.

GRUBER, T. R. Toward principles for the design of ontologies used for knowledge sharing? **International Journal of Human-Computer Studies**, [S.l.], v. 43, n. 5, p. 907–928, 1995.

ROLSTADÅS, A. (Ed.). The Role of Competency Questions in Enterprise Engineering. In: _____. **Benchmarking — Theory and Practice**. Boston, MA: Springer US, 1995. p. 22–31.

GUARINO, N. **Formal ontology in information systems**: proceedings of the first international conference (fois'98), june 6-8, trento, italy. [S.l.]: IOS press, 1998. v. 46.

GUO, J.; ZHANG, L. Convex set reliability-based optimal attitude control for uncertain dynamic systems. **ISA Transactions**, [S.l.], v. 99, p. 371–379, 2020.

GUO, X.; WANG, Z.; ZHANG, C.; ZHANG, H.; HUANG, C. Dual-mode robust fuzzy model predictive control of time-varying delayed uncertain nonlinear systems with perturbations. **IEEE Transactions on Fuzzy Systems**, [S.l.], v. 31, n. 7, p. 2182–2196, 2023.

GUPTA, A.; AGARWAL, S. NCDT-CSS: enhancing performance using noncoherent distributed transmission of chirp spread spectrum. **IEEE Internet of Things Journal**, [S.l.], v. 12, n. 7, p. 7969–7979, 2025.

GUPTA, N.; SHARMA, V. Context Aware Hybrid Network Architecture for Iot with Machine Learning Based Intelligent Gateway. **SN Computer Science**, [S.l.], v. 4, n. 3, p. 297, 2023.

GUSENBAUER, M.; HADDAWAY, N. R. Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources. **Research Synthesis Methods**, [S.l.], v. 11, n. 2, p. 181–217, 2020.

HABBAL, A.; ALI, M. K.; ABUZARAIDA, M. A. Artificial Intelligence Trust, Risk and Security Management (AI TRiSM): frameworks, applications, challenges and future research directions. **Expert Systems with Applications**, [S.l.], v. 240, p. 122442, 2024.

HAMEED, A. et al. A machine learning regression approach for throughput estimation in an IoT environment. In: IEEE ITHINGS/GREENCOM/CPSCOM/SMARTDATA/CYBERMATICS, 2021. **Proceedings...** [S.l.: s.n.], 2021. p. 29–36.

HAMEED, A.; VIOLOS, J.; SANTI, N.; LEIVADEAS, A.; MITTON, N. A Machine Learning Regression Approach for Throughput Estimation in an IoT Environment. , [S.l.], p. 29–36, 2021.

HAO, J.; BOUZOUANE, A.; GABOURY, S. An incremental learning method based on formal concept analysis for pattern recognition in nonstationary sensor-based smart environments. **Pervasive and Mobile Computing**, [S.l.], v. 59, p. 101045, 2019.

HARIS, I.; BISANOVIC, V.; WALLY, B.; RAUSCH, T.; RATASICH, D.; DUSTDAR, S.; KAPPEL, G.; GROSU, R. Sensyml: simulation environment for large-scale iot applications. In: IECON 2019 - 45TH ANNUAL CONFERENCE OF THE IEEE INDUSTRIAL ELECTRONICS SOCIETY, 2019. **Anais...** [S.l.: s.n.], 2019. v. 1, p. 3024–3030.

HECKLER, W. F.; CARVALHO, J. V. de; BARBOSA, J. L. V. Machine learning for suicidal ideation identification: a systematic literature review. **Computers in Human Behavior**, [S.l.], v. 128, p. 107095, 2022.

HENNING, S.; HASSELBRING, W. Scalable and reliable multi-dimensional aggregation of sensor data streams. In: IEEE INTERNATIONAL CONFERENCE ON BIG DATA (BIG DATA), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p. 3512–3517.

HORROCKS, I.; PATEL-SCHNEIDER, P. F.; BOLEY, H.; TABET, S.; GROSOF, B.; DEAN, M. SWRL: a semantic web rule language combining owl and ruleml. , [S.l.], 2004. Available at: `https://www.w3.org/Submission/SWRL/`. Access at: 10 dez. 2024.

HOSEINY, F.; AZIZI, S.; SHOJAFAR, M.; TAFAZOLLI, R. Joint QoS-aware and Cost-efficient Task Scheduling for Fog-cloud Resources in a Volunteer Computing System. , New York, NY, USA, v. 21, n. 4, 2021.

HOU, J.; LU, H.; NAYAK, A. A GNN-based proactive caching strategy in NDN networks. **Peer-to-Peer Networking and Applications**, [S.l.], v. 16, n. 2, p. 997–1009, 2023.

HU, C.; SUN, Z.; LI, C.; ZHANG, Y.; XING, C. Survey of Time Series Data Generation in IoT. **Sensors**, [S.l.], v. 23, n. 15, p. 6976, 2023.

HUANG, B.; LIU, X.; XIANG, Y.; YU, D.; DENG, S.; WANG, S. Reinforcement learning for cost-effective IoT service caching at the edge. **Journal of Parallel and Distributed Computing**, [S.l.], v. 168, p. 120–136, 2022.

HUSSAIN, D. I.; ELOMRI, D. A.; KERBACHE, D. L.; OMRI, D. A. E. Smart city solutions: comparative analysis of waste management models in iot-enabled environments using multiagent simulation. **Sustainable Cities and Society**, [S.l.], v. 103, p. 105247, 2024.

IDRIS, S.; KARUNATHILAKE, T.; FöRSTER, A. Survey and Comparative Study of LoRa-Enabled Simulators for Internet of Things and Wireless Sensor Networks. **Sensors**, [S.l.], v. 22, n. 15, 2022.

IMRAN; IQBAL, N.; KIM, D. H. IoT Task Management Mechanism Based on Predictive Optimization for Efficient Energy Consumption in Smart Residential Buildings. **Energy and Buildings**, [S.l.], v. 257, p. 111762, 2022.

ISABONA, J.; IMOIZE, A. L.; KIM, Y. Machine Learning-Based Boosted Regression Ensemble Combined with Hyperparameter Tuning for Optimal Adaptive Learning. **Sensors**, [S.l.], v. 22, n. 10, 2022.

ISOLANI, P. H.; HAXHIBEQIRI, J.; MOERMAN, I.; HOEBEKE, J.; GRANVILLE, L. Z.; LATRÉ, S.; MARQUEZ-BARJA, J. M. Sd-ran interactive management using in-band network telemetry in ieee 802.11 networks. **Journal of Network and Systems Management**, [S.l.], v. 31, n. 1, p. 11, 2023.

JAMSHIDI, S.; AMIRNIA, A.; NIKANJAM, A.; NAFI, K. W.; KHOMH, F.; KEIVANPOUR, S. Self-adaptive cyber defense for sustainable IoT: a drl-based ids optimizing security and energy efficiency. **Journal of Network and Computer Applications**, [S.l.], v. 239, p. 104176, 2025.

JANSSEN, M.; KUK, G. Digital transformation of government: towards a citizen-centric and ethical approach. **Global Policy**, [S.l.], 2024.

JASKIERNY, L.; KOTULSKI, L. A self-adapting IoT network configuration supported by distributed graph transformations. **Appl. Sci.**, [S.l.], v. 13, n. 23, p. 12718, 2023.

JAYARAM, R.; PRABAKARAN, S. Adaptive cost and energy aware secure peer-to-peer computational offloading in the edge-cloud enabled healthcare system. **Peer-to-Peer Networking and Applications**, [S.l.], v. 14, n. 4, p. 2209–2223, 2021.

JUAN, A. A.; KEENAN, P.; MARTí, R.; MCGARRAGHY, S.; PANADERO, J.; CARROLL, P.; OLIVA, D. A review of the role of heuristics in stochastic optimisation: from metaheuristics to learnheuristics. **Annals of Operations Research**, [S.l.], v. 320, n. 2, p. 831–861, 2023.

KARUNKUZHALI, D.; MEENAKSHI, B.; LINGAM, K. An adaptive fuzzy c means with seagull optimization algorithm for analysis of WSNs in agricultural field with IoT. **Wireless Personal Communications**, [S.l.], v. 126, n. 2, p. 1459–1480, 2022.

KAUR, M.; ARON, R. An energy-efficient load balancing approach for scientific workflows in fog computing. **Wireless Personal Communications**, [S.l.], v. 125, n. 4, p. 3549–3573, 2022.

KESHAV, S. How to read a paper. **ACM SIGCOMM Computer Communication Review**, [S.l.], v. 37, n. 3, p. 1–2, 2016.

KHAN, A. A.; YANG, J.; LAGHARI, A. A.; BAQASAH, A. M.; ALROOBAEA, R.; KU, C. S.; ALIZADEHSANI, R.; ACHARYA, U. R.; POR, L. Y. BAIoT-EMS: consortium network for small-medium enterprises management system with blockchain and augmented intelligence of things. **Engineering Applications of Artificial Intelligence**, [S.l.], v. 141, p. 109838, 2025.

KHAN, A.; ALI, S.; HAYAT, S.; AZEEM, M.; ZHONG, Y.; ZAHID, M. A.; ALENAZI, M. J. Fault-tolerance and unique identification of vertices and edges in a graph: the fault-tolerant mixed metric dimension. **Journal of Parallel and Distributed Computing**, [S.l.], v. 197, p. 105024, 2025.

KHAN, S.; KHAN, S.; ALI, Y.; KHALID, M.; ULLAH, Z.; MUMTAZ, S. Highly accurate and reliable wireless network slicing in 5th generation networks: a hybrid deep learning approach. **Journal of Network and Systems Management**, [S.l.], v. 30, n. 2, p. 29, 2022.

KHIATI, M.; DJENOURI, D. Adaptive learning-enforced broadcast policy for solar energy harvesting wireless sensor networks. **Computer Networks**, [S.l.], v. 143, p. 263–274, 2018.

KIM, B.-S.; KIM, K.-I.; SHAH, B.; CHOW, F.; KIM, K. H. Wireless Sensor Networks for Big Data Systems. **Sensors**, [S.l.], v. 19, n. 7, 2019.

KOHYARNEJADFARD, I.; ALOISE, D.; AZHARI, S. V.; DAGENAIS, M. R. Anomaly detection in microservice environments using distributed tracing data analysis and NLP. **Journal of Cloud Computing**, [S.l.], v. 11, n. 1, p. 25, 2022.

KORKALAINEN, M.; SALLINEN, M.; KäRKKäINEN, N.; TUKEVA, P. Survey of Wireless Sensor Networks Simulation Tools for Demanding Applications. In: FIFTH INTERNATIONAL CONFERENCE ON NETWORKING AND SERVICES, 2009., 2009. **Anais...** [S.l.: s.n.], 2009. p. 102–106.

LAB, I. B. R. **Intel Lab Data Dataset**. 2004.

LAKHAN, A.; MEMON, M. S.; MASTOI, Q.-u.-a.; ELHOSENY, M.; MOHAMMED, M. A.; QABULIO, M.; ABDEL-BASSET, M. Cost-efficient mobility offloading and task scheduling for microservices IoVT applications in container-based fog cloud network. **Cluster Computing**, [S.l.], p. 1–23, 2022.

LALOTRA, G. S.; KUMAR, V.; BHATT, A.; CHEN, T.; MAHMUD, M. iReTADS: an intelligent real-time anomaly detection system for cloud communications using temporal data summarization and neural network. **Security and Communication Networks**, [S.l.], n. 1, p. 9149164, 2022.

LATEEF HAROON P S, A.; PATIL, S. N.; BIDARE DIVAKARACHARI, P.; FALKOWSKI-GILSKI, P.; RAFEEQ, M. D. An optimized system for sensor ontology meta-matching using swarm intelligent algorithm. **Internet Technology Letters**, [S.l.], v. 7, n. 4, p. e498, 2024.

LATHA, R.; John Justin Thangaraj, D. S. IoT security using heuristic aided symmetric convolution-based deep temporal convolution network for intrusion detection by extracting multi-cascaded deep attention features. **Expert Systems with Applications**, [S.l.], v. 269, p. 126363, 2025.

LI, C.; ZHANG, Y.; LUO, Y. Deep reinforcement learning-based resource allocation and seamless handover in multi-access edge computing based on SDN. **Knowledge and Information Systems**, [S.l.], v. 63, n. 9, p. 2479–2511, 2021.

LI, G.; ZHANG, T.; TSAI, C.-Y.; YAO, L.; LU, Y.; TANG, J. Review of the metaheuristic algorithms in applications: visual analysis based on bibliometrics. **Expert Systems with Applications**, [S.l.], v. 255, p. 124857, 2024.

LI, X.; ZHAO, H.; FENG, Y.; LI, J.; ZHAO, Y.; WANG, X. Research on key technologies of high energy efficiency and low power consumption of new data acquisition equipment of power Internet of Things based on artificial intelligence. **International Journal of Thermofluids**, [S.l.], v. 21, p. 100575, 2024.

LI, Z.; YANG, H.; SUN, C. Multi-objective optimization-inspired set theory-based regularization approach for uncertain systems. **Applied Soft Computing**, [S.l.], v. 95, p. 106515, 2020.

LIU, D. et al. Sensors anomaly detection of industrial internet of things based on isolated forest algorithm and data compression. **Scientific Programming**, [S.l.], v. 2021, p. 6699313, 2021.

LIU, D.; ZHEN, H.; KONG, D.; CHEN, X.; ZHANG, L.; YUAN, M.; WANG, H. Sensors Anomaly Detection of Industrial Internet of Things Based on Isolated Forest Algorithm and Data Compression. **Scientific Programming**, [S.l.], v. 2021, n. 1, p. 6699313, 2021.

LIU, F.; XIALIANG, T.; YUAN, M.; LIN, X.; LUO, F.; WANG, Z.; LU, Z.; ZHANG, Q. Evolution of Heuristics: towards efficient automatic algorithm design using large language model. In: FORTY-FIRST INTERNATIONAL CONFERENCE ON MACHINE LEARNING, 2024. **Anais...** [S.l.: s.n.], 2024.

LOGESHWARAN, J.; SHANMUGASUNDARAM, N.; LLORET, J. Energy-efficient resource allocation model for device-to-device communication in 5G wireless personal area networks. **International Journal of Communication Systems**, [S.l.], v. 36, n. 13, p. e5524, 2024.

MADHUNALA, S.; ANANTHA, B. Centralized Monitored Spectrum Management using Multi-resource Parallel Sensing in Cognitive Radio Networks. , New York, NY, USA, 2022.

MALEKZADEH, M. Performance prediction and enhancement of 5G networks based on linear regression machine learning. **EURASIP Journal on Wireless Communications and Networking**, [S.l.], v. 2023, n. 1, p. 74, 2023.

MANFREDI, V.; WOLFE, A. P.; ZHANG, X.; WANG, B. Learning an Adaptive Forwarding Strategy for Mobile Wireless Networks: resource usage vs. latency. **arXiv preprint arXiv:2207.11386**, [S.l.], 2022.

MARIć, J.; BACH, M. P.; GUPTA, S. The origins of digital service innovation (DSI): systematic review of ontology and future research agenda. **Journal of Service Management**, [S.l.], v. 35, n. 2, p. 141–175, 2024.

MARKUS, A.; KECSKEMETI, G.; KERTESZ, A. Flexible Representation of IoT Sensors for Cloud Simulators. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p. 199–203.

MATATHAMMAL, A.; GUPTA, K.; LAVANYA, L.; HALGATTI, A. V.; GUPTA, P.; VAIDHYANATHAN, K. EdgeMLBalancer: a self-adaptive approach for dynamic model switching on resource-constrained edge devices. In: IEEE 22ND INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE COMPANION (ICSA-C), 2025., 2025. **Anais...** [S.l.: s.n.], 2025. p. 543–552.

AL MUBARAK, M.; HAMDAN, A. (Ed.). A Systematic Review of IoT Forensics-Based on a Permissioned Blockchain. In: _____. **Innovative and Intelligent Digital Technologies; Towards an Increased Efficiency**: volume 2. Cham: Springer Nature Switzerland, 2024. p. 341–350.

MEENA, V.; KRITHIVASAN, K.; RAHUL, P.; PRABA, T. S. Toward an intelligent cache management: in an edge computing era for delay sensitive iot applications. **Wireless Personal Communications**, [S.l.], v. 131, n. 2, p. 1075–1088, 2023.

METWALLY, K. M.; JARRAY, A.; KARMOUCH, A. Two-phase ontology-based resource allocation approach for IaaS cloud service. In: ANNUAL IEEE CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE (CCNC), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 790–795.

MIRDULA, S.; ROOPA, M. MUD enabled deep learning framework for anomaly detection in IoT integrated smart building. **e-Prime - Advances in Electrical Engineering, Electronics and Energy**, [S.l.], v. 5, p. 100186, 2023.

MOCNEJ, J.; SEAH, W. K.; PEKAR, A.; ZOLOTOVA, I. Decentralised IoT Architecture for Efficient Resources Utilisation. **IFAC-PapersOnLine**, [S.l.], v. 51, n. 6, p. 168–173, 2018. 15th IFAC Conference on Programmable Devices and Embedded Systems PDeS 2018.

MORABITO, R. Virtualization on Internet of Things Edge Devices with Container Technologies: a performance evaluation. **IEEE Access**, [S.l.], v. 5, p. 8835–8850, 2017.

MORAES, J.; OLIVEIRA, H.; CERQUEIRA, E.; BOTH, C.; ZEADALLY, S.; ROSÁRIO, D. Evaluation of an adaptive resource allocation for lorawan. **Journal of Signal Processing Systems**, [S.l.], v. 94, n. 1, p. 65–79, 2022.

MO'TAZ, A.-H.; MAABREH, M.; TAAMNEH, S.; PRADEEP, A.; SALAMEH, H. B. Apache Hadoop performance evaluation with resources monitoring tools, and parameters optimization: iot emerging demand. **Journal of Theoretical and Applied Information Technology**, [S.l.], v. 99, n. 11, 2021. Accessed in: 25 Oct 2024.

MUNIR, K.; Sheraz Anjum, M. The use of ontologies for effective knowledge modelling and information retrieval. **Applied Computing and Informatics**, [S.l.], v. 14, n. 2, p. 116–126, 2018.

MUNISWAMY, S.; VIGNESH, R. DSTS: a hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment. **Journal of Cloud Computing**, [S.l.], v. 11, n. 1, p. 33, 2022.

MUSEN, M. A. The protégé project: a look back and a look forward. **AI Matters**, New York, NY, USA, v. 1, n. 4, p. 4–12, June 2015.

MUSEN, M. **Protégé 5.5.0**. Available at: `https://protege.stanford.edu/`. Accessed at: 10 dez. 2024.

NAGARAJAN, S.; RANI, P. S.; VINMATHI, M. S.; SUBBA REDDY, V.; SALETH, A. L. M.; ABDUS SUBHAHAN, D. Multi agent deep reinforcement learning for resource allocation in container-based clouds environments. **Expert Systems**, [S.l.], 2023.

NANDISH, B.; PUSHPARAJESH, V. Efficient power management based on adaptive whale optimization technique for residential load. **Electrical Engineering**, [S.l.], p. 1–18, 2024.

NANDYALA, C. S.; KIM, H.-W.; CHO, H.-S. QTAR: a q-learning-based topology-aware routing protocol for underwater wireless sensor networks. **Computer Networks**, [S.l.], v. 222, p. 109562, 2023.

NAZARI, A.; KORDABADI, M.; MOHAMMADI, R.; LAL, C. EQRSRL: an energy-aware and qos-based routing schema using reinforcement learning in iomt. **Wireless Networks**, [S.l.], v. 29, n. 7, p. 3239–3253, 2023.

NOETZOLD, D. **SensorSimulator**. Accessed on: 22 Dec. 2024.

NOY, N. F.; MCGUINNESS, D. L. et al. **Ontology development 101**: a guide to creating your first ontology. [S.l.]: Stanford knowledge systems laboratory technical report KSL-01-05 and . . . , 2001.

NOY, N.; MCGUINNESS, D. **Ontology Development 101**: a guide to creating your first ontology. 2001. v. 32.

NúñEZ, A.; CAñIZARES, P. C.; de Lara, J. CloudExpert: an intelligent system for selecting cloud system simulators. **Expert Systems with Applications**, [S.l.], v. 187, p. 115955, 2022.

O'CONNOR, M. J.; DAS, A. K. Supporting Rule Systems on the Semantic Web with SWRL. In: INTERNATIONAL SEMANTIC WEB CONFERENCE, 2005. **Proceedings...** [S.l.: s.n.], 2005. p. 97–110.

OSMAN, R. A. Optimizing IoT communication for enhanced data transmission in smart farming ecosystems. **Expert Systems with Applications**, [S.l.], v. 265, p. 125879, 2025.

PADGHAM, L.; WINIKOFF, M. **Developing Intelligent Agent Systems**: a practical guide. [S.l.]: John Wiley & Sons Ltd., 2004.

PENG, Y.; WU, I.-C. A cloud-based monitoring system for performance analysis in IoT industry. **The Journal of Supercomputing**, [S.l.], v. 77, n. 8, p. 9266–9289, 2021.

PENG, Z.; SONG, X.; SONG, S.; STOJANOVIC, V. Spatiotemporal fault estimation for switched nonlinear reaction–diffusion systems via adaptive iterative learning. **International Journal of Adaptive Control and Signal Processing**, [S.l.], v. 38, n. 10, p. 3473–3483, 2024.

PéREZ, J.; ARENAS, M.; GUTIERREZ, C. Semantics and complexity of SPARQL. **ACM Trans. Database Syst.**, New York, NY, USA, v. 34, n. 3, Sept. 2009.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: EASE), 12., 2008. **Anais...** [S.l.: s.n.], 2008.

PFLANZNER, T.; KERTESZ, A.; SPINNEWYN, B.; LATRé, S. MobIoTSim: towards a mobile iot device simulator. In: IEEE 4TH INTERNATIONAL CONFERENCE ON FUTURE INTERNET OF THINGS AND CLOUD WORKSHOPS (FICLOUDW), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 21–27.

POLLEY, J.; BLAZAKIS, D.; MCGEE, J.; RUSK, D.; BARAS, J. ATEMU: a fine-grained sensor network simulator. In: FIRST ANNUAL IEEE COMMUNICATIONS SOCIETY CONFERENCE ON SENSOR AND AD HOC COMMUNICATIONS AND NETWORKS, 2004. IEEE SECON 2004., 2004., 2004. **Anais...** [S.l.: s.n.], 2004. p. 145–152.

PRABHU, D.; ALAGESWARAN, R.; MIRUNA JOE AMALI, S. Multiple agent based reinforcement learning for energy efficient routing in WSN. **Wireless Networks**, [S.l.], v. 29, n. 4, p. 1787–1797, 2023.

PRAMOD KUMAR, P.; SAGAR, K. Reinforcement learning and neuro-fuzzy GNN-based vertical handover decision on internet of vehicles. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 35, n. 12, p. e7688, 2023.

PRASANNA, B.; RAMYA, D.; SHELKE, N.; FERNANDES, J. B.; GALETY, M. G.; ASHOK, M. Radial basis function neural network-based algorithm unfolding for energy-aware resource allocation in wireless networks. **Wireless Networks**, [S.l.], p. 1–18, 2023.

PRIYA, S. A.; BHAT, N.; KANNA, B. R.; RAJALAKSHMI, S.; JEYAVATHANA, R. B.; S, S. Proactive Network Optimization Using Deep Learning in Predicting IoT Traffic Dynamics. In: INTERNATIONAL CONFERENCE ON INNOVATIVE PRACTICES IN TECHNOLOGY AND MANAGEMENT (ICIPTM), 2024., 2024. **Anais...** [S.l.: s.n.], 2024. p. 1–6.

PRIYA, S. A. et al. Proactive network optimization using deep learning in predicting IoT traffic dynamics. In: ICIPTM, 4., 2024, Noida, India. **Proceedings...** [S.l.: s.n.], 2024. p. 1–6.

PUIU, D.; BARNAGHI, P. et al. CityPulse: real-time iot data streams for urban intelligence. **Sensors**, [S.l.], v. 16, n. 9, p. 1513, 2024.

PUTRA, M. A. P.; HERMAWAN, A. P.; KIM, D.-S.; LEE, J.-M. Data Prediction-Based Energy-Efficient Architecture for Industrial IoT. **IEEE Sensors Journal**, [S.l.], v. 23, n. 14, p. 15856–15866, 2023.

RAIBULET, C.; OH, J.; LEEST, J. Analysis of MAPE-K loop in self-adaptive systems for cloud, IoT and CPS. In: SERVICE-ORIENTED COMPUTING – ICSOC 2022 WORKSHOPS, 2023. **Anais...** Springer, 2023. p. 130–141.

RAJ, R. S.; HEMA, L. K. Dynamic clustering optimization for energy efficient IoT Network: a simple constrastive graph approach. **Expert Systems with Applications**, [S.l.], v. 264, p. 125875, 2025.

RAMKUMAR, J.; VADIVEL, R. Multi-adaptive routing protocol for internet of things based ad-hoc networks. **Wireless Personal Communications**, [S.l.], v. 120, n. 2, p. 887–909, 2021.

RAO, C. K.; SAHOO, S. K.; YANINE, F. F. An IoT Enabled Energy Management System with Precise Forecasting and Load Optimization for PV Power Generation. **Transactions of the Indian National Academy of Engineering**, [S.l.], p. 1–21, 2024.

RATH, C. K.; MANDAL, A. K.; SARKAR, A. Dynamic provisioning of devices in microservices-based IoT applications using context-aware reinforcement learning. **Innovations in Systems and Software Engineering**, [S.l.], p. 1–14, 2024.

REDHEFFER, R. M. The Riccati equation: initial values and inequalities. **Mathematische Annalen**, [S.l.], v. 133, p. 235–250, 1957. Received 22 October 1956.

REHMAN, A.; ALBLUSHI, I. G. M.; ZIA, M. F.; KHALID, H. M.; INAYAT, U.; BENBOUZID, M.; MUYEEN, S.; HUSSAIN, G. A. A solar-powered multi-functional portable charging device (SPMFPCD) with internet-of-things (IoT)-based real-time monitoring—An innovative scheme towards energy access and management. **Green Technologies and Sustainability**, [S.l.], v. 3, n. 1, p. 100134, 2025.

RESTUCCIA, F.; MELODIA, T. DeepWiERL: bringing deep reinforcement learning to the internet of self-adaptive things. In: IEEE INFOCOM, 2020. **Proceedings...** [S.l.: s.n.], 2020. p. 844–853.

REVANESH, M.; GUNDAL, S. S.; ARUNKUMAR, J.; JOSEPHSON, P. J.; SUHASINI, S.; DEVI, T. K. Artificial neural networks-based improved Levenberg–Marquardt neural network for energy efficiency and anomaly detection in WSN. **Wireless Networks**, [S.l.], p. 1–16, 2023.

ROSSETTO, A. G. d. M.; NOETZOLD, D.; SILVA, L. A.; LEITHARDT, V. R. Q. Enhancing Monitoring Performance: a microservices approach to monitoring with spyware techniques and prediction models. **Sensors**, [S.l.], v. 24, n. 13, 2024.

RTX, U. **Machine Failure Prediction Using Sensor Data**. Available at: `https://www.kaggle.com/datasets/umerrtx/machine-failure-prediction-using-sensor-data`, Accessed on: February 11, 2025.

RZEVSKI, G.; SKOBELEV, P.; ZHILYAEV, A. Emergent Intelligence in Smart Ecosystems: conflicts resolution by reaching consensus in resource management. **Mathematics**, [S.l.], v. 10, n. 11, 2022.

S, R.; KANNIGA, D. An Innovation of Distributed Scheduling and QoS Localized Routing Scheme for Wireless Industrial Sensor Network. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND ELECTRICAL CIRCUITS AND ELECTRONICS (ICDCECE), 2023., 2023. **Anais...** [S.l.: s.n.], 2023. p. 1–6.

SAH, D. K. et al. Load-balance scheduling for intelligent sensors deployment in industrial internet of things. **Cluster Computing**, [S.l.], v. 25, p. 1715–1727, 2022.

SAH, D. K.; NGUYEN, T. N.; CENGIZ, K.; DUMBA, B.; KUMAR, V. Load-balance scheduling for intelligent sensors deployment in industrial internet of things. **Cluster Computing**, [S.l.], v. 25, n. 3, p. 1715–1727, 2022.

SAKR, H.; ELSABROUTY, M. Meta-reinforcement learning for edge caching in vehicular networks. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], v. 14, n. 4, p. 4607–4619, 2023.

SAMARAKOON, S.; BANDARA, S.; JAYASANKA, N.; HETTIARACHCHI, C. Self-Healing and Self-Adaptive Management for IoT-Edge Computing Infrastructure. In:

MORATUWA ENGINEERING RESEARCH CONFERENCE (MERCON), 2023., 2023. **Anais...** [S.l.: s.n.], 2023. p. 473–478.

SAMARAKOON, S. et al. Self-healing and self-adaptive management for IoT-edge computing infrastructure. In: MERCON, 2023, Moratuwa, Sri Lanka. **Proceedings...** [S.l.: s.n.], 2023. p. 473–478.

SANCHEZ, L.; GALACHE, J. **SmartSantander**: a real-world iot testbed for smart cities. 2024.

SANGEETHA, S.; LOGESHWARAN, J.; FAHEEM, M.; KANNADASAN, R.; SUNDARARAJU, S.; VIJAYARAJA, L. Smart performance optimization of energy-aware scheduling model for resource sharing in 5G green communication systems. **The Journal of Engineering**, [S.l.], n. 2, p. e12358, 2024.

SARITHA, K.; SARASVATHI, V. An Energy-Efficient and QoS-Preserving Hybrid Cross-Layer Protocol Design for Deep Learning-Based Air Quality Monitoring and Prediction. , Berlin, Heidelberg, v. 5, n. 3, 2024.

SATER, R. A.; HAMZA, A. B. A Federated Learning Approach to Anomaly Detection in Smart Buildings. , New York, NY, USA, v. 2, n. 4, 2021.

SAVAGLIO, C.; FORTINO, G. A Simulation-driven Methodology for IoT Data Mining Based on Edge Computing. **ACM Trans. Internet Technol.**, New York, NY, USA, v. 21, n. 2, Mar. 2021.

SELIM, G. E. I.; HEMDAN, E. E.-D.; SHEHATA, A. M.; EL-FISHAWY, N. A. Anomaly events classification and detection system in critical industrial internet of things infrastructure using machine learning algorithms. **Multimedia Tools and Applications**, [S.l.], v. 80, n. 8, p. 12619–12640, 2021.

SELVARAJAN, S.; MANOHARAN, H.; AL-SHEHARI, T.; ALSADHAN, N. A.; SINGH, S. IoT driven healthcare monitoring with evolutionary optimization and game theory. **Scientific Reports**, [S.l.], v. 15, n. 1, p. 15224, 2025.

SENNAN, S.; RAMASUBBAREDDY, S.; DHANARAJ, R. K.; NAYYAR, A.; BALUSAMY, B. Energy-efficient cluster head selection in wireless sensor networks-based internet of things (IoT) using fuzzy-based Harris hawks optimization. **Telecommunication Systems**, [S.l.], p. 1–17, 2024.

SHAFIEE, M.; OZEV, S. An In-Field Programmable Adaptive CMOS LNA for Intelligent IoT Sensor Node Applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v. 41, n. 2, p. 201–210, 2022.

SHARMA, D. K.; RODRIGUES, J. J.; VASHISHTH, V.; KHANNA, A.; CHHABRA, A. RLProph: a dynamic programming based reinforcement learning approach for optimal routing in opportunistic iot networks. **Wireless Networks**, [S.l.], v. 26, p. 4319–4338, 2020.

SHARMA, M.; KAUR, P. XLAAM: explainable lstm-based activity and anomaly monitoring in a fog environment. **Journal of Reliable Intelligent Environments**, [S.l.], v. 9, n. 4, p. 463–477, 2023.

SHARMA, N.; MANGLA, M.; MOHANTY, S. N.; GUPTA, D.; TIWARI, P.; SHORFUZZAMAN, M.; RAWASHDEH, M. A smart ontology-based IoT framework for remote patient monitoring. **Biomedical Signal Processing and Control**, [S.l.], v. 68, p. 102717, 2021.

SHARMA, S.; SINGH, B. Context aware autonomous resource selection and Q-learning based power control strategy for enhanced cooperative awareness in LTE-V2V communication. **Wireless Networks**, [S.l.], v. 26, p. 4045–4060, 2020.

SHI, J.; ZHU, G.; LI, H.; HAWBANI, A.; WU, J.; LIN, N.; ZHAO, L. Multi-AAVs flocking for navigation and obstacle avoidance in network-constrained environments. **IEEE Internet of Things Journal**, [S.l.], v. 12, n. 7, p. 8931–8946, 2025.

SHIRMARZ, A.; GHAFFARI, A. Automatic software defined network (SDN) performance management using TOPSIS decision-making algorithm. **Journal of Grid Computing**, [S.l.], v. 19, n. 2, p. 16, 2021.

SHUKLA, A.; SINGH, D.; SAJWAN, M.; VERMA, A.; KUMAR, A. A source location privacy preservation scheme in WSN-assisted IoT network by randomized ring and confounding transmission. **Wireless Networks**, [S.l.], v. 28, n. 2, p. 827–852, 2022.

SHUKRY, S.; FAHMY, Y. Traffic load access barring scheme for random-access channel in massive machine-to-machine and human-to-human devices coexistence in LTE-A. **International Journal of Communication Systems**, [S.l.], v. 34, n. 8, p. e4777, 2024. e4777 dac.4777.

SIMAIYA, S. et al. Hybrid CNN–LSTM deep learning model optimized by evolutionary algorithms for cloud resource utilization prediction. **Scientific Reports**, [S.l.], v. 14, n. 1, p. 3204, 2024.

SINGH, G.; CHATURVEDI, A. K. A cost, time, energy-aware workflow scheduling using adaptive PSO algorithm in a cloud–fog environment. **Computing**, [S.l.], p. 1–30, 2024.

SINGH, K.; MALHOTRA, J. Fuzzy link cost estimation based adaptive tree algorithm for routing optimization in wireless sensor networks using reinforcement learning. **International Journal of Sensors Wireless Communications and Control**, [S.l.], v. 8, n. 3, p. 151–164, 2018.

SINGH, R.; SHARMA, R.; Vaseem Akram, S.; GEHLOT, A.; BUDDHI, D.; MALIK, P. K.; ARYA, R. Highway 4.0: digitalization of highways for vulnerable road safety development with intelligent iot sensors and machine learning. **Safety Science**, [S.l.], v. 143, p. 105407, 2021.

SOBIERAJ, M.; KOTYńSKI, D. Docker performance evaluation across operating systems. **Applied Sciences**, [S.l.], v. 14, n. 15, p. 6672, 2024.

SOMESULA, M. K.; KOTTE, A.; ANNADANAM, S. C.; MOTHKU, S. K. Deadline-aware cache placement scheme using fuzzy reinforcement learning in device-to-device mobile edge networks. **Mobile Networks and Applications**, [S.l.], v. 27, n. 5, p. 2100–2117, 2022.

SOROUR, S. E.; ALJAAFARI, M.; SHAKER, A. M.; AMIN, A. E. LSTM-JSO framework for privacy preserving adaptive intrusion detection in federated IoT networks. **Scientific Reports**, [S.l.], v. 15, n. 1, p. 11321, 2025.

SORRENTINO, M.; FRANZESE, N.; TRIFIRÒ, A. Development and experimental assessment of a multi-annual energy monitoring tool to support energy intelligence and management in telecommunication industry. **Energy Efficiency**, [S.l.], v. 17, n. 6, p. 64, 2024.

SOUNDARI, A. G.; JYOTHI, V. Energy efficient machine learning technique for smart data collection in wireless sensor networks. **Circuits, Systems, and Signal Processing**, [S.l.], v. 39, n. 2, p. 1089–1122, 2020.

SOURI, A.; HUSSIEN, A.; HOSEYNINEZHAD, M.; NOROUZI, M. A systematic review of IoT communication strategies for an efficient smart environment. **Transactions on Emerging Telecommunications Technologies**, [S.l.], v. 33, n. 3, p. e3736, 2022.

SRICHANDAN, S. K.; MAJHI, S. K.; JENA, S.; MISHRA, K.; RAO, D. C. Efficient latency-and-energy-aware IoT-fog-cloud task orchestration: novel algorithmic approach with enhanced arithmetic optimization and pattern search. **International Journal of Information Technology**, [S.l.], v. 16, n. 5, p. 3311–3324, 2024.

STEHLE, F. K.; VANDELLI, W.; ZAHN, F.; AVOLIO, G.; FR{oNING, H. DeepHYDRA: a hybrid deep learning and dbscan-based approach to time-series anomaly detection in dynamically-configured systems. In: 2024, New York, NY, USA. **Anais...** Association for Computing Machinery, 2024.

STEIN, C. M.; ROCKENBACH, D. A.; GRIEBLER, D.; TORQUATI, M.; MENCAGLI, G.; DANELUTTO, M.; FERNANDES, L. G. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 33, n. 11, p. e5786, 2020.

STEPHAN, T.; AL-TURJMAN, F.; K, S. J.; BALUSAMY, B. Energy and spectrum aware unequal clustering with deep learning based primary user classification in cognitive radio sensor networks. **International Journal of Machine Learning and Cybernetics**, [S.l.], v. 12, n. 11, p. 3261–3294, 2021.

STEPHAN, T.; SHARMA, K.; SHANKAR, A.; PUNITHA, S.; VARADARAJAN, V.; LIU, P. Fuzzy-logic-inspired zone-based clustering algorithm for wireless sensor networks. **International Journal of Fuzzy Systems**, [S.l.], v. 23, p. 506–517, 2021.

SUBRAMANIAN, M.; NARAYANAN, M.; BHASKER, B.; GNANAVEL, S.; HABIBUR RAHMAN, M.; PRADEEP REDDY, C. H. Hybrid Electro Search with Ant Colony Optimization Algorithm for Task Scheduling in a Sensor Cloud Environment for Agriculture Irrigation Control System. **Complexity**, [S.l.], n. 1, p. 4525220, 2022.

SULIMANI, H.; SAJJAD, A. M.; ALGHAMDI, W. Y.; KAIWARTYA, O.; JAN, T.; SIMOFF, S.; PRASAD, M. Reinforcement optimization for decentralized service placement policy in IoT-centric fog environment. **Transactions on Emerging Telecommunications Technologies**, [S.l.], v. 34, n. 11, p. e4650, 2023.

SUNDARESAN, Y. B.; DURAI, M. S. A high performance cognitive framework (SIVA-self intelligent versatile and adaptive) for heterogenous architecture in IOT environment. **International Journal of Reasoning-based Intelligent Systems**, [S.l.], v. 10, n. 3-4, p. 269–278, 2018.

SURESHKUMAR, C.; SABENA, S. Design of an adaptive framework with compressive sensing for spatial data in wireless sensor networks. **Wireless Networks**, [S.l.], v. 29, n. 5, p. 2203–2216, 2023.

SURYAWAN, I. G. T.; PUTRA, I. K. N.; MELIANA, P. M.; SUDIPA, I. G. I. Performance Comparison of ARIMA, LSTM, and Prophet Methods in Sales Forecasting. **Sinkron : jurnal dan penelitian teknik informatika**, [S.l.], v. 8, n. 4, p. 2410–2421, Oct. 2024.

SUSAN SHINY, G.; MUTHU KUMAR, B. E2IA-HWSN: energy efficient dual intelligent agents based data gathering and emergency event delivery in heterogeneous wsn enabled iot. **Wireless Personal Communications**, [S.l.], v. 122, p. 379–408, 2022.

SUSHA, I.; GIL-GARCIA, J. R. Governing AI in cities: challenges and emerging approaches. **Technovation**, [S.l.], v. 124, p. 102717, 2023.

TALAAT, F. M. Effective deep Q-networks (EDQN) strategy for resource allocation based on optimized reinforcement learning algorithm. **Multimedia Tools and Applications**, [S.l.], v. 81, n. 28, p. 39945–39961, 2022.

TALAAT, F. M.; SARAYA, M. S.; SALEH, A. I.; ALI, H. A.; ALI, S. H. A load balancing and optimization strategy (LBOS) using reinforcement learning in fog computing environment. **Journal of Ambient Intelligence and Humanized Computing**, [S.l.], v. 11, n. 11, p. 4951–4966, 2020.

TAM, P.; MATH, S.; KIM, S. Priority-aware resource management for adaptive service function chaining in real-time intelligent IoT services. **Electronics**, [S.l.], v. 11, n. 19, p. 2976, 2022.

TAM, P.; MATH, S.; KIM, S. Priority-aware resource management for adaptive service function chaining in real-time intelligent IoT services. **Electronics**, [S.l.], v. 11, n. 19, p. 2976, 2022.

TRUONG, H.-L.; FAHRINGER, T.; NERIERI, F.; DUSTDAR, S. Performance metrics and ontology for describing performance data of grid workflows. In: CCGRID 2005. IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2005., 2005. **Anais...** [S.l.: s.n.], 2005. v. 1, p. 301–308 Vol. 1.

ULLAH, I.; KIM, C.-M.; HEO, J.-S.; HAN, Y.-H. An energy-efficient data collection scheme by mobile element based on Markov decision process for wireless sensor networks. **Wireless Personal Communications**, [S.l.], v. 123, n. 3, p. 2283–2299, 2022.

VAIDHYANATHAN, K.; CAPORUSCIO, M.; FLORIO, S.; MUCCINI, H. ML-enabled Service Discovery for Microservice Architecture: a qos approach. In: ACM/SIGAPP SYMPOSIUM ON APPLIED COMPUTING, 39., 2024, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2024. p. 1193–1200. (SAC '24).

VELRAJAN, S.; CERONMANI SHARMILA, V. QoS-Aware Service Migration in Multi-access Edge Compute Using Closed-Loop Adaptive Particle Swarm Optimization Algorithm. , USA, v. 31, n. 1, 2022.

VELRAJAN, S.; SHARMILA, V. C. QoS-aware service migration in multi-access edge computing using closed-loop adaptive particle swarm optimization algorithm. **Journal of Network and Systems Management**, [S.l.], v. 31, n. 1, p. 17, 2023.

VIANNA, H. D.; BARBOSA, J. L. V. In search of computer-aided social support in non-communicable diseases care. **Telematics and Informatics**, [S.l.], v. 34, n. 8, p. 1419–1432, 2017.

VINJAMURI, U. R.; RAO, B. L. Efficient energy management system using Internet of things with FORDF technique for distribution system. **IET Renewable Power Generation**, [S.l.], v. 15, n. 3, p. 676–688, 2021.

WANG, B.; FAN, T.-y.; NIE, X.-t. Advanced delay assured numerical heuristic modelling for peer to peer project management in cloud assisted internet of things platform. **Peer-to-Peer Networking and Applications**, [S.l.], v. 13, n. 6, p. 2166–2176, 2020.

WANG, C.-C. A comparison study between fuzzy time series model and ARIMA model for forecasting Taiwan export. **Expert Systems with Applications**, [S.l.], v. 38, n. 8, p. 9296–9304, 2011.

WANG, Z.; LAM, H.-K.; GUO, Y.; XIAO, B.; LI, Y.; SU, X.; YEATMAN, E. M.; BURDET, E. Adaptive event-triggered control for nonlinear systems with asymmetric state constraints: a prescribed-time approach. **IEEE Transactions on Automatic Control**, [S.l.], v. 68, n. 6, p. 3625–3632, 2023.

WEERASINGHE, S.; ZASLAVSKY, A.; LOKE, S. W.; MEDVEDEV, A.; ABKEN, A.; HASSANI, A.; HUANG, G.-L. Reinforcement Learning Based Approaches to Adaptive Context Caching in Distributed Context Management Systems. , New York, NY, USA, v. 5, n. 2, 2024.

WEI, B.; XIE, Z.; LIU, Y.; WEN, K.; DENG, F.; ZHANG, P. Online Monitoring Method for Insulator Self-explosion Based on Edge Computing and Deep Learning. **CSEE Journal of Power and Energy Systems**, [S.l.], v. 8, n. 6, p. 1684–1696, 2022.

WU, J.; ZHANG, G.; NIE, J.; PENG, Y.; ZHANG, Y. Deep Reinforcement Learning for Scheduling in an Edge Computing-Based Industrial Internet of Things. **Wireless Communications and Mobile Computing**, [S.l.], n. 1, p. 8017334, 2021.

WU, Q.; WANG, S.; GE, H.; FAN, P.; FAN, Q.; LETAIEF, K. B. Delay-sensitive task offloading in vehicular fog computing-assisted platoons. **IEEE Transactions on Network and Service Management**, [S.l.], 2023.

XU, H.; LIU, W.-d.; LI, L.; YAO, D.-j.; MA, L. FSRW: fuzzy logic-based whale optimization algorithm for trust-aware routing in iot-based healthcare. **Scientific Reports**, [S.l.], v. 14, n. 1, p. 16640, 2024.

XU, M.; BUYYA, R. BrownoutCon: a software system based on brownout and containers for energy-efficient cloud computing. **Journal of Systems and Software**, [S.l.], v. 155, p. 91–103, 2019.

XU, R.; HUANG, Z.; CHEN, S.; LI, J.; WU, P.; LIN, Y. Wi-CL: low-cost wifi-based detection system for nonmotorized traffic travel mode classification. **Journal of Advanced Transportation**, [S.l.], n. 1, p. 1033717, 2023.

XU, X.; LIU, N.; PAN, Z. Distributed Reinforcement Learning for Optimizing Age of Information and Energy Consumption in Wireless Powered IoT Systems. In: 2023, New York, NY, USA. **Anais...** Association for Computing Machinery, 2023.

XUE, J.; WU, S.; JI, Z.; PAN, W. Research on Intelligent Server Room Integrated Operation and Maintenance Management System. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND COMPUTER INFORMATION TECHNOLOGY (AICIT), 2023., 2023. **Anais...** [S.l.: s.n.], 2023. p. 1–6.

YANG, Z.; ABBASI, I. A.; MUSTAFA, E. E.; ALI, S.; ZHANG, M. An Anomaly Detection Algorithm Selection Service for IoT Stream Data Based on Tsfresh Tool and Genetic Algorithm. **Security and Communication Networks**, [S.l.], n. 1, p. 6677027, 2021.

YANG, Z. et al. An anomaly detection algorithm selection service for IoT stream data based on Tsfresh tool and genetic algorithm. **Security and Communication Networks**, [S.l.], v. 2021, p. Article ID 6677027, 2021.

YASEEN, A.; MALY, K. J.; ZEIL, S. J.; ZUBAIR, M. Performance Evaluation of Oracle Semantic Technologies with Respect to User Defined Rules. In: INTERNATIONAL WORKSHOP ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 2011., 2011. **Anais...** [S.l.: s.n.], 2011. p. 252–256.

YOU, D.; LIN, W.; SHI, F.; LI, J.; QI, D.; FONG, S. A novel approach for CPU load prediction of cloud server combining denoising and error correction. **Computing**, [S.l.], p. 1–18, 2023.

YOUSEFI, S.; DERAKHSHAN, F.; KARIMIPOUR, H.; AGHDASI, H. S. An efficient route planning model for mobile agents on the internet of things using Markov decision process. **Ad Hoc Networks**, [S.l.], v. 98, p. 102053, 2020.

ZANELLA GOMES, J.; VICTóRIA BARBOSA, J. L.; RESIN GEYER, C. F.; ANJOS, J. C. Santos dos; VICENTE CANTO, J.; PESSIN, G. Ubiquitous Intelligent Services for Vehicular Users: a systematic mapping. **Interacting with Computers**, [S.l.], v. 31, n. 5, p. 465–479, 11 2019.

ZARE, N.; MACIOSZEK, E.; GRANà, A.; GIUFFRè, T. Blending Efficiency and Resilience in the Performance Assessment of Urban Intersections: a novel heuristic informed by literature review. **Sustainability**, [S.l.], v. 16, n. 6, 2024.

ZESHAN, F.; AHMAD, A.; BABAR, M. I.; HAMID, M.; HAJJEJ, F.; ASHRAF, M. An IoT-Enabled Ontology-Based Intelligent Healthcare Framework for Remote Patient Monitoring. **IEEE Access**, [S.l.], v. 11, p. 3947–3960, 2023.

ZESHAN, F.; AHMAD, A.; BABAR, M. I.; HAMID, M.; HAJJEJ, F.; ASHRAF, M. An IoT-enabled ontology-based intelligent healthcare framework for remote patient monitoring. **IEEE Access**, [S.l.], v. 11, p. 133947–133966, 2023.

ZHANG, K.; LIU, Y.; WANG, X.; MEI, F.; SUN, G.; ZHANG, J. Enhancing IoT (Internet of Things) feature selection: a two-stage approach via an improved whale optimization algorithm. **Expert Systems with Applications**, [S.l.], v. 256, p. 124936, 2024.

ZHAO, H.; WANG, H.; NIU, B.; ZHAO, X.; ALHARBI, K. H. Event-triggered fault-tolerant control for input-constrained nonlinear systems with mismatched disturbances via adaptive dynamic programming. **Neural Networks**, [S.l.], v. 164, p. 508–520, 2023.

ZHENG, T.; WAN, J.; ZHANG, J.; JIANG, C. Deep reinforcement learning-based workload scheduling for edge computing. **Journal of Cloud Computing**, [S.l.], v. 11, n. 1, p. 3, 2022.

ZHOU, H. A novel approach to cloud resource management: hybrid machine learning and task scheduling. **Journal of Grid Computing**, [S.l.], v. 21, n. 4, p. 68, 2023.

ZHOU, Y.; LI, B.; WANG, H. Reliability-constrained uncertain spacecraft sliding mode attitude control using interval techniques. **Aerospace Science and Technology**, [S.l.], v. 112, p. 106620, 2021.

ZHUANG, D.; GAN, V. J.; TEKLER, Z. D.; CHONG, A.; TIAN, S.; SHI, X. Data-driven predictive control for smart HVAC system in IoT-integrated buildings with time-series forecasting and reinforcement learning. **Applied Energy**, [S.l.], v. 338, p. 120936, 2023.

ZOLFAGHARI, M.; GHOLAMI, S. A hybrid approach of adaptive wavelet transform, long short-term memory and ARIMA-GARCH family models for the stock index prediction. **Expert Systems with Applications**, [S.l.], v. 182, p. 115149, 2021.