www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(4), 2026, 41-56. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2026.v17i4.1296

Dynamic Load Balancing and Fault Tolerance for Machine Learning Deployments Using a Process Control Table

Cesar Primero-Huerta[1,2], Luis-Armando Guadarrama-Osorio[1], Mariana-Carolyn Cruz-Mendoza[1] & Eddy Sánchez-DeLaCruz[2]*

[1] División de Ingeniería en Sistemas Computacionales, Tecnológico Nacional de México-Tecnológico de Estudios Superiores de Valle de Bravo.
[2] Artificial Intelligence Lab., Tecnológico Nacional de México-Instituto Tecnológico Superior de Misantla.
Email address(es): iscprimerocesar@gmail.com, jesusgma33@gmail.com, marianacaroloyn@hotmail.com, eddsacx@gmail.com
✉Corresponding author: *Eddy Sánchez-DeLaCruz*

Abstract. Deploying machine learning models in production environments presents significant challenges when systems must process high request volumes while maintaining stable real-time performance. Existing solutions often address request distribution, performance monitoring, and fault management separately, which may contribute to bottlenecks, increased latency, and service saturation. This study presents the design of a Process Control Table that acts as a central coordinator for orchestrating the real-time execution of machine learning models. The proposed mechanism dynamically routes incoming tasks to the least occupied worker available at the time of assignment. Prototype testing indicates that the Process Control Table reduces latency and maintains more stable response times under high demand than the traditional schemes evaluated in the study, which tended to become saturated. The mechanism also promotes equitable workload distribution and incorporates automatic failover capabilities to improve deployment reliability. These findings indicate that centralised process coordination can support dynamic load balancing, request optimisation, and fault tolerance in machine learning deployments.
Keywords: machine learning deployment; MLOps; dynamic load balancing; fault tolerance; Process Control Table; model serving; request routing; worker scheduling; latency optimisation; automatic failover; distributed machine learning systems; real-time model execution.

Article information

Received: Jan 17, 2026
Accepted: May 19, 2026

1 Introduction

The use of Machine Learning (ML) has expanded rapidly beyond the academic domain, becoming a key component of industrial and business applications. This expansion is evident across a wide range of fields, from the real-time optimization of chemical processes (Zhang et al., 2019) to predictive monitoring in smart manufacturing (Aliev & Antonelli, 2021; Puranik & Mahmood, 2025), and even the enhancement of cybersecurity through intrusion detection systems (Mora, 2024). However, a significant gap still exists between the development of a functional model in the laboratory and its successful deployment in production environments (Walker, 2023). Bridging this gap is essential, since the true value of ML lies not only in experimentation, but primarily in its reliable and scalable operation under real-world production conditions (Walker, 2023).

Two major challenges arise in the implementation of ML models, a process commonly referred to as MLOps (Machine Learning Operations). The first challenge concerns real-time execution management: production models must be capable of handling unpredictable and highly variable inference workloads. Without an appropriate management mechanism, concurrent requests may overwhelm server resources, generating

© Editorial Académica Dragón Azteca S. de R.L. de C.V. (EDITADA.ORG), All rights reserved.

bottlenecks that lead to increased latency and potential system failures (Kamila et al., 2022; Walker, 2023). The second challenge involves lifecycle management. A deployed model, once made available to external applications, is not static; rather, it must be continuously monitored to detect performance degradation, a phenomenon known as model drift (de la Rúa Martínez, 2020; Emma, 2025). Conventional monitoring systems, which are often based on batch processing, typically operate with substantial delays that may extend to several hours, making them unsuitable for applications that require real-time response or detection capabilities (de la Rúa Martínez, 2020).

A review of the current literature, presented in Section 2, shows that existing studies tend to address ML deployment challenges in a fragmented manner. On the one hand, some studies focus on optimizing load distribution in cloud environments (Shafiq et al., 2022; Vashistha et al., 2025) and on improving reliability and fault recovery capabilities (Kamila et al., 2022). On the other hand, other works concentrate on MLOps-specific monitoring for detecting model performance degradation, commonly referred to as drift (de la Rúa Martínez, 2020), or on the real-time visualization of key performance indicators (KPIs), often from an industrial perspective (Moens et al., 2020; Puranik & Mahmood, 2025). However, an important limitation remains: the absence of an integrated architectural design that combines three essential functions—dynamic load distribution, real-time status monitoring, and model lifecycle management—under a centralized and efficient control mechanism.

To address this limitation, this study proposes the design and implementation of a process control table. The proposal is grounded in concurrency control principles, which represent a major challenge in large-scale computing environments (Walker, 2023). Efficient management of simultaneous requests is fundamental to load balancing and is also critical for ensuring fault tolerance and high performance (Kamila et al., 2022; Shafiq et al., 2022).

The main objective of this research is to design and validate a modular architecture centered on a process control table for the intelligent management of ML model execution and monitoring. Specifically, this study seeks to answer the following research questions: (1) How can a modular architecture centered on a control table be designed to simultaneously manage request concurrency and ML model lifecycle status? and (2) To what extent does this design improve response latency and load distribution compared with a static deployment system?

The methodology adopted in this research was experimental. A functional prototype was designed and implemented by integrating key components, including the Control Module and containerized workers (executors). To ensure the applicability of the results to critical production environments, the experimental design replicates a high-frequency financial fraud detection scenario, which serves as a standard benchmark for validating latency-sensitive MLOps architectures. The prototype was evaluated through load testing to measure and validate its performance under high-demand conditions.

This study differs from previous work by substantially extending the scope and technical depth of the SMaDE proceedings version. In particular, the present manuscript provides a more comprehensive description of the proposed architecture by explicitly detailing the interaction among the User, the Gateway, and the Redis-backed Control Table during a prediction request, as well as the Fault Tolerance and State Consistency Flow. In addition, this version incorporates new evaluation metrics to characterize system performance more precisely, strengthens the validation of fault tolerance by explaining the internal logic of the Control Module, and clarifies how worker exceptions are handled while preserving the semaphore pattern through the finally block.

Furthermore, unlike the preliminary proceedings paper, this manuscript explicitly discusses the study's limitations and enhances reproducibility by making the configuration and testing package publicly available. Therefore, the current article should be understood as an extended and methodologically more robust version of the earlier SMaDE publication.

The remainder of this paper is organized as follows. Section 2 reviews related work and the state of the art. Section 3 details the design methodology, the proposed system architecture, and the structure of the control table. Section 4 presents and discusses the performance test results. Finally, Section 5 summarizes the conclusions and outlines future work.

2 Related Work

The main objective is the design of a system for executing and monitoring ML models in real time involves the integration of several key fields: ML operations (MLOps), cloud load balancing, system monitoring, and software architectures (de la Rúa Martínez, 2020; Emma, 2025; Jin & Yang, 2025). Below, we review previous research in these areas, which serves as the foundation for this research.

2.1 MLOps and Deployment Challenges

Migrating an ML model from a laboratory environment to a large-scale production system represents a major challenge for organizations, as it involves complex issues related to infrastructure, inference performance, and data management (Walker, 2023). The practice of MLOps[1] seeks to apply DevOps principles to automate and standardize the entire ML lifecycle (Emma, 2025). A particularly relevant approach for modern container-based applications has been discussed by Elgamal et al. (2025). The main challenges associated with large-scale deployment include the need for infrastructure that can adapt to changing demand, the management of data processing bottlenecks, the optimization of inference latency, and the automation of repetitive tasks (Maddireddy et al., 2025; Walker, 2023). To achieve effective MLOps, lifecycle automation and efficient load balancing are considered fundamental (Jin & Yang, 2025; Walker, 2023).

Beyond the initial deployment, a key aspect of MLOps is continuous monitoring to identify model drift, which refers to performance degradation caused by changes in real-world data. Such monitoring is essential to ensure the reliability and sustained performance of the model over time, and it has become a fundamental pillar of data and model management in modern cloud platforms (de la Rúa Martínez, 2020; Emma, 2025; Yao et al., 2025).

2.2 Load Balancing for Inference

Managing the real-time execution of machine learning models is closely associated with the system's ability to handle unpredictable and fluctuating volumes of inference requests (Jin & Yang, 2025). Load balancing is responsible for distributing these requests across multiple servers or workers[2] in order to prevent overload and reduce latency (Shafiq et al., 2022). In general, load balancing strategies are classified as static (e.g., round-robin) or dynamic (i.e., based on the current system load). Dynamic methods, which make decisions according to the system state, are considered more effective for handling variable workloads in cloud environments (Low et al., 2024; Shafiq et al., 2022).

To overcome the limitations of dynamic methods, which react only to the instantaneous state of the system, current research trends are oriented toward more intelligent balancing approaches capable of predicting and proactively managing load allocation (Sliwko, 2024; Yao et al., 2021). These approaches include the use of deep learning techniques to model and anticipate system workloads, thereby optimizing task assignment (Sanjalawe et al., 2025; Vashistha et al., 2025). In addition, centralized traffic management through software-defined networking (SDN) enables more flexible resource orchestration (Mahdizadeh et al., 2024). These balancing principles are essential not only for the high-performance execution of inference workloads (Jin & Yang, 2025), but also for AI training workloads (McClure et al., 2025).

[1] DevOps (Development Operations) is a culture that integrates software development (Dev) and operations (Ops) to automate and accelerate software delivery, through practices like Continuous Integration (CI) and Continuous Delivery (CD) (Elgamal et al., 2025; Emma, 2025).
[2] A service instance designed to receive units of work.

In this study, real-time monitoring is not approached merely as the supervision of variables, but rather as a continuous process of data acquisition, processing, and inference aimed at supporting decision-making. The workflow comprises: (i) acquisition of operational data, (ii) preprocessing and validation, (iii) extraction of relevant features, (iv) evaluation using the predictive model, and (v) generation of alerts for preventive intervention.

2.3 Real-time Monitoring and Predictive Maintenance

A significant challenge in traditional MLOps systems is monitoring latency. Many of these systems operate in batch mode, which means that model drift detection may take several hours. For critical applications, however, real-time monitoring is essential. In this context, de la Rúa Martínez (2020) proposed a streaming architecture based on Spark and Kafka that is capable of detecting drift within seconds. This highlights the importance of immediate processing for the monitoring and visualization of key performance indicators (KPIs) (Moens et al., 2020; Puranik & Mahmood, 2025).

The concept of immediate KPI visualization is also fundamental in industrial environments, particularly within the Industrial Internet of Things (IIoT[3]). For instance, Puranik and Mahmood (2025) implemented a real-time monitoring system that uses Grafana for instant operational monitoring and Power BI for historical analysis. Similarly, Moens et al. (2020) designed a three-tier architecture to monitor machine fleets and generate data to support predictive maintenance.

2.4 Proactive Systems and Fault Tolerance

Real-time monitoring provides the foundation for designing proactive systems capable of anticipating failures. Several studies have applied ML to predict faults from sensor data. Aliev and Antonelli (2021) used ML to analyze wear in collaborative robots (cobots) and predict downtime. Similarly, Ibadov et al. (2025) employed Random Forest models with integrated sensor data to estimate the remaining useful life of components with high accuracy. This predictive approach also extends to cloud infrastructure. From a reactive failure management perspective, Kamila et al. (2022) showed that applying ML techniques, such as XGBoost, to performance metrics (e.g., CPU and memory usage) enables the prediction of service outages and accelerates recovery by 38.15%. The use of AI to develop predictive failure models has become a central strategy in highly reliable microservices-based systems (Yalamati, 2025). The feasibility of such systems also depends on software-level optimizations, including process improvement through neural networks (Zhang et al., 2019), the implementation of efficient control primitives in ML frameworks (Yu et al., 2018), and their integration into security dashboards (Mora, 2024).

From the analysis of these studies, a central limitation emerges: the literature provides effective solutions for MLOps (Elgamal et al., 2025; Emma, 2025; Walker, 2023), load balancing (Jin & Yang, 2025; Sanjalawe et al., 2025; Shafiq et al., 2022), and real-time monitoring (de la Rúa Martínez, 2020; Kamila et al., 2022; Puranik & Mahmood, 2025), but these are generally proposed as independent systems. There is still a lack of architectures that rely on a single, centralized, low-latency data structure to coordinate these three functions simultaneously: dynamic load balancing, real-time monitoring, and model lifecycle management (e.g., updates or self-correction).

This research addresses this gap through the design of a Process Control Table that coordinates the three aforementioned functions. The table acts as a centralized state repository and a single coordination point for the real-time execution (Maddireddy et al., 2025; Yu et al., 2018), load distribution (Jin & Yang, 2025; Shafiq et al., 2022), and monitoring (de la Rúa Martínez, 2020; Puranik & Mahmood, 2025) of ML models.

[3] Industrial Internet of Things (IoT) is the application of the Internet of Things (IoT) to the industrial sector (SAP, 2025).

The analysis conducted in this work strengthens the acquisition and processing stage and establishes the conditions under which the predictive model can operate in a timely and reliable manner.

3 Methodology

This section describes the implementation of the proposed methodology, detailing the system architecture, the processing workflow, and the technical components that enable its operation. It also explains the relationships among modules and the design decisions adopted to ensure consistency with the objective of the study.

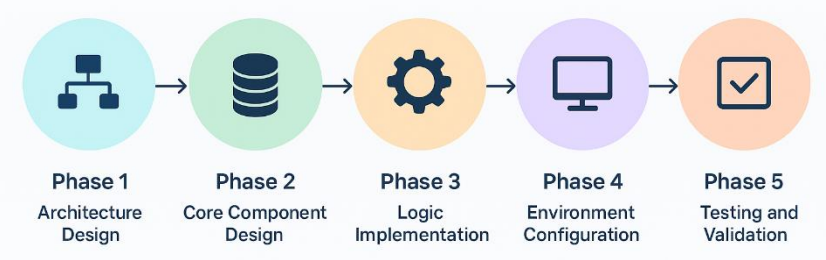

Fig. 1. Methodology used.

3.1 Phase 1: Architecture Design

In the first phase, we defined a design referred to as the Control-Centered ML Execution Architecture (AECC). This design follows a modular approach, inspired by three-tier architectures used for monitoring machine fleets (Moens et al., 2020), and organizes the system into four main components. Figure 2 illustrates the integrated interaction among these components. The process begins with data input, which feeds the processing module. Subsequently, the transformed data are sent to the model where a prediction is generated. The output is then transferred to visualization, while the observed events are reintegrated as feedback to update the system and improve its performance.

Entry Point (API Gateway). This acts as the single gateway for all external prediction requests that arrive at the system via HTTP. Its main function is to receive these requests and redirect them internally to the Control Module. It functions as a facade that simplifies interaction for clients. To ensure system integrity, the Gateway acts as a Policy Enforcement Point (PEP). Authentication (AuthN) is implemented via the validation of JSON Web Tokens (JWT) signed by a trusted Identity Provider (OpenID Connect OIDC), ensuring that only legitimate clients with valid credentials can establish a connection. Authorization (AuthZ) is enforced through Role-Based Access Control (RBAC) and token scopes (e.g., model:predict or admin:view); the Gateway inspects the token claims to verify if the authenticated client has the specific permissions required to access the requested model. Furthermore, it implements stability patterns such as circuit-breaking and backpressure. If the Control Module reports saturation or repeated timeouts, the Gateway preemptively rejects excess requests (returning HTTP 503) to prevent cascading failures in the internal infrastructure.

Control Module (Brain). This is the central component that coordinates the AECC. It does not execute ML models directly, but instead acts as an intelligent manager of request traffic. It contains the Load Manager logic, which queries the Process Control Table to obtain updated information on the status and load of each worker. Based on this information, it decides which specific worker each new request should be sent to, following the defined balancing algorithm (see Phase 3). It maintains a direct, low-latency connection with the Control Table.

Model Executors (Workers). These are the computing units that actually perform the ML predictions (inferences). Each worker is an independent instance, typically packaged as a microservice in a Docker container, running a specific version of an ML model. This design allows multiple instances of the same model (or different models) to operate simultaneously. It also facilitates adaptation to demand, as workers can be added or removed as needed to handle the volume of requests without interrupting service.

Monitoring Module. This component is responsible for observing the overall system status in real time. It operates in parallel, reading information contained in the Process Control Table (such as the current load of each worker, its status, and average latency) and, optionally, directly querying the workers' status. It uses this data to update visualization panels (dashboards), such as those that can be built with Grafana (Puranik & Mahmood, 2025), offering a clear view of system performance. It also plays a key role in reliability management, as it can detect inactive or faulty workers (based on the last_activity field or repeated failures) and take corrective actions, such as marking a worker as unavailable in the Control Table, similar to proactive approaches for predicting failures (Kamila et al., 2022).

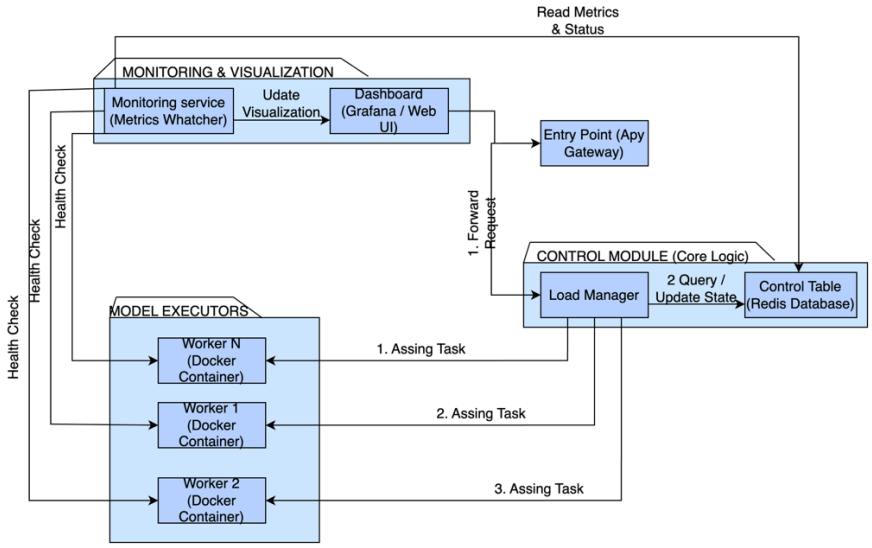

Fig. 2. AECC architecture diagram. Shows the flow of a request from the API Gateway, through the Control Module (which queries the Control Table) and being directed to one of the Model Executors. The Monitoring Module reads the table for the dashboard.

3.2 Phase 2: Process Control Table Design

As the central component of the proposed architecture, the design of the control table constituted a critical phase. A key-value data structure optimized for high-speed operations was defined. To ensure minimal latency, the table was implemented in an in-memory database using Redis. With respect to operational overhead, the system employs pipelined operations to maintain the impact of read/write latency below 0.5 ms per request, thereby ensuring that the control mechanism does not become a bottleneck. Regarding resilience, although the current prototype uses a standalone Redis instance to isolate algorithmic performance, the architectural design explicitly decouples the control logic from the storage topology. The connection interface of the Control Module is implemented through standard drivers compatible with Redis Sentinel and Cluster modes. This design choice ensures that migration to a High Availability (HA) configuration requires only configuration changes rather than code refactoring, thereby effectively mitigating the Single Point of Failure (SPOF) risk inherent in centralized control structures.

Furthermore, a heartbeat mechanism was implemented through the last_activity field, which functions as a Time-to-Live (TTL) strategy. Workers update this timestamp periodically (e.g., every 5 seconds); if the difference between the current time and last_activity exceeds a predefined threshold, the Monitoring Module automatically marks the worker as inactive, thereby handling node failures without manual intervention. The table structure (see Table 1) was designed to store the complete real-time status of every active worker in the system. Each field was selected to fulfill a specific role within the control and monitoring logic.

Table 1. Structure of the Process Control Table

Field	Example Value	Purpose
id_worker	worker-uuid-001	Unique executor ID.
id_model	modelo fraude	Model it is running.
version	1.2	Model version.
status	ACTIVE	ACTIVE,INACTIVE,ERROR.
endpoint	”http://10.0.1.5:80”	Worker IP/port address.
current_requests	3	Current request counter
avg_latency_ms	150.4	Average latency (KPI) (Moens *et al.,* 2020).
last_activity	(Timestamp)	To detect inactive workers.

3.3 Phase 3: Control Logic Implementation

With the architecture and table defined, the management algorithms that use this table were designed. A dynamic load balancing algorithm based on the Least Connections technique was implemented, considered superior to static approaches (Shafiq *et al.,* 2022). The workflow for each incoming request is as follows:

1) A new request for modelo_fraude arrives at the Control Module.
2) The Load Manager queries Table 1 and filters by id_model = 'modelo_fraude' and status = 'ACTIVE'.
3) It obtains a list of available workers (e.g., W1, W2, W3).
4) Routing Decision applies the logic, selecting the worker with the minimum value in the current_requests field.
5) The update increments the selected worker's (e.g., W2) current_requests counter in the table and forwards the request to its endpoint.
6) When worker W2 responds, its counter is decremented.

Logic was designed for the Monitoring Module and lifecycle management, both dependent on the control table (Kamila et al., 2022).

- Monitoring (Read): A process using Grafana (Puranik & Mahmood, 2025) is defined to query the table every second. It aggregates data to display real-time KPIs, such as total load (SUM(current_requests)) and average latency (AVG(avg_latency_ms)) per model.
- Auto-correction (Write): The Monitoring Module also acts as a watchdog. If it detects that a worker has not updated its last_activity in a while (e.g., 60 seconds) or if it fails repeatedly, it can change its status to ERROR. The Load Manager will automatically stop sending it traffic.
- Model Updates: A flow for zero-downtime updates was designed. To deploy version 1.3 of a model, a new worker (W4) is launched and registered in the table with status = 'INACTIVE'. After a test, its status is changed to ACTIVE. Simultaneously, the old workers (1.2) are changed to UPDATING. The manager stops sending them new requests, and when their current_requests counter reaches 0, they are shut down and removed from the table.
- Drift Detection Pipeline: The architecture extends beyond simply observing performance to active model lifecycle management. To detect model drift, the Monitoring Module performs a systematic check using the Kolmogorov-Smirnov (KS) test on a continuous stream of incoming data. This test compares the distribution of features within a sliding window (e.g., the last 1,000 requests) against the model's original training data. The KS test calculates the difference between these two data distributions. If the calculated probability (p) of the two distributions being the same falls below a strict limit ($p<0.05$), the system recognizes a significant statistical change. This triggers an automated event via a message queue signal to launch the retraining pipeline. Using a message queue guarantees the request is processed, separating the monitoring task from the heavy retraining process and improving overall system resilience.

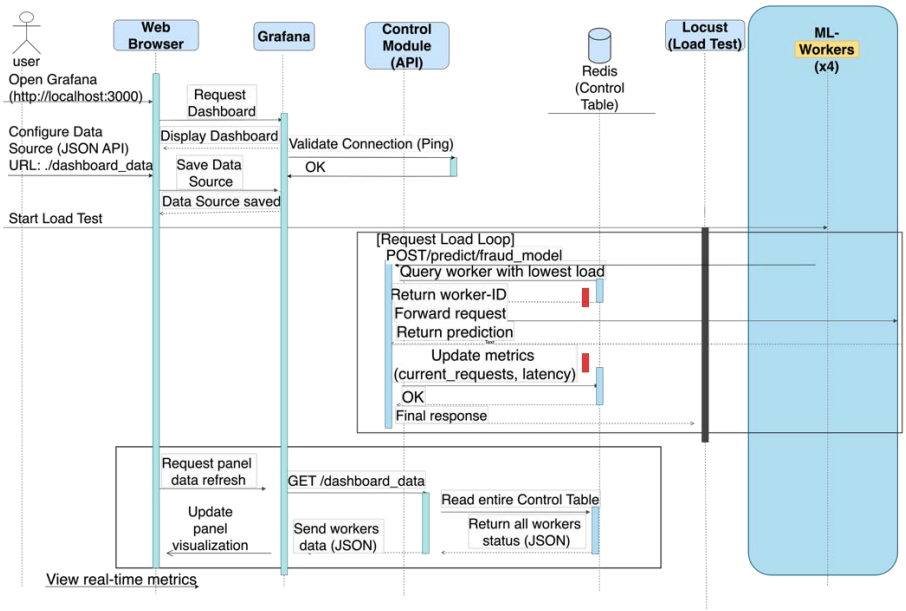

Fig. 3. Sequence diagram of the real-time monitoring and prediction flow.

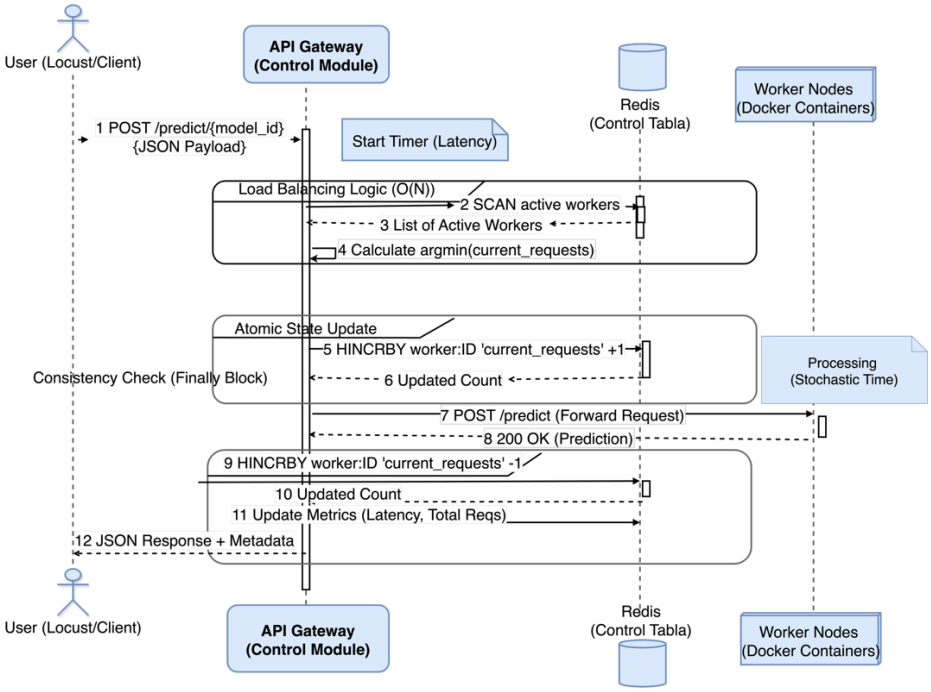

Fig. 4. End-to-End Sequence Diagram. Illustrates the interaction between the User, the Gateway, and the Redis-backed Control Table during a prediction request.

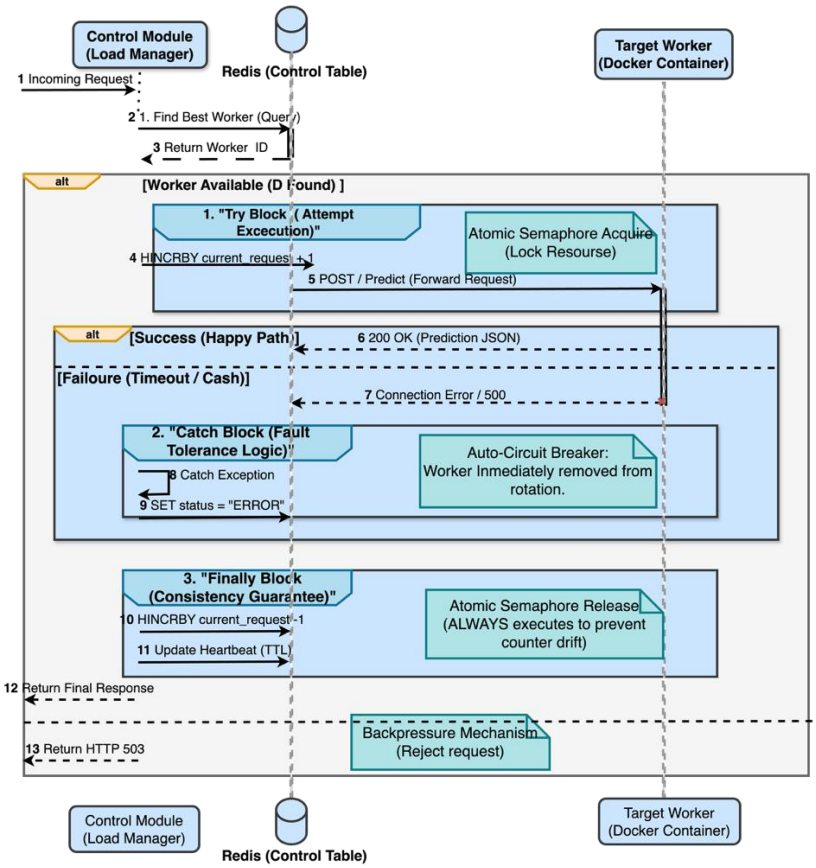

Fig. 5. Fault Tolerance and State Consistency Flow. Details the internal logic within the Control Module, specifically how the system handles worker failures (exceptions) and enforces the semaphore pattern via the finally block.

3.4 Algorithmic Formalization of the Control Strategy

To rigorously define the load balancing strategy, let $W = \{w_1, w_2, \dots, w_n\}$ be the set of available worker nodes registered in the Control Table. Each worker w_i is characterized by a tuple state S_i

$$S_i = (status_i, C_i, T_{last}) \quad (1)$$

Where $status_i \in \{ACTIVE, INACTIVE, ERROR\}$, C_i represents the current concurrent requests count (current_requests), and T_{last} is the timestamp of the last heartbeat.
The Load Manager implements a minimization function f(W) to select the optimal target worker w^* for an incoming request r_t at time t:

$$w^* = \arg\min_{w_i \in W} (C_i) \quad (2)$$

Subject to the following operational constraints:

$$status_i = ACTIVE \quad (3)$$

$$(t - T_{last}) < T_{timeout} \quad (4)$$

Where $T_{timeout}$ is the maximum allowable silence period before a worker is considered failed. In the event of a tie where multiple workers share the same minimum C_i, the system selects the worker with the lowest index i.

3.5 Phase 4: Experimental Environment Setup

To validate the AECC architecture, a functional prototype was implemented. This phase consisted of selecting the technological tools to build each component. Python with the FastAPI framework was used for the Control Module and Workers, chosen for its high performance in asynchronous operations (async/await), ideal for managing multiple non-blocking I/O requests. Redis was selected as the in-memory database for creating the Process Control Table, given its sub-millisecond latency and native atomic operations (like INCR and DECR), which are fundamental for managing request counters (current_requests) without race conditions. For the Execution Environment, the ML workers were designed to be packaged as Docker containers, allowing for isolation and scalability.

3.6 Phase 5: Test and Validation Plan

The final phase of the methodology was to define the experimental plan to validate the prototype's performance. Locust, a Python-based load testing tool, was used to simulate a staggered increase of concurrent users. The validation plan focused on comparing two scenarios:

Scenario A (Static). A standard deployment with a static load balancer (Round Robin type).

Scenario B (Dynamic). Our prototype implementing the AECC architecture with the Control Table.

The following key metrics to be collected were defined: average response latency (ms), which measures the total time from when the user sends the request until they receive the response. Next, we have request distribution, which counts how many requests each individual worker processed in a batch of 10,000 requests. Finally, fault tolerance, which observes the system's behavior when simulating a worker failure.

4 Results and Discussion

This section presents the evaluation of the proposed system and the analysis of the results obtained. It describes the test scenarios, the indicators considered, and the interpretation of system performance, with the aim of assessing the effectiveness of the proposed methodological approach.

4.1 Computational Complexity and Overhead Analysis

A critical aspect of the design is the computational overhead introduced by the Control Module. Since Redis operations such as HINCRBY and HGETALL operate with a time complexity of O(1), the total overhead per request is dominated by the linear scan of registered workers used to find the minimum load.

If N is the number of active workers, the complexity of the routing decision is O(N). For the experimental cluster size ($N = 4$), the retrieval time is negligible (< 0.5 ms). However, the architecture is designed to scale. For massive scaling scenarios ($N > 1000$), the look-up complexity can be reduced to O(log N) by implementing a heap-based priority queue within Redis sorted sets (ZSET), although this remains future work.

Distributed Semaphore Pattern: The implementation acts as a distributed semaphore system (as shown in the sequence diagrams). By atomically incrementing the current_requests counter before forwarding the request and ensuring its decrement within a finally block (regardless of the inference outcome), the system guarantees state consistency. This mechanism effectively implements Application-Layer Backpressure. Unlike network-layer load balancers that rely on TCP queue metrics, our approach is aware of the semantic state of the application, preventing the ”Thundering Herd” problem where a recovering node is immediately overwhelmed by pending requests.

4.2 Experimental Setup

The experiments were conducted on a controlled testbed using Docker containers to minimize external interference. The infrastructure consisted of a cluster with 4 worker nodes, each allocated with 2 vCPUs and 4GB of RAM, running on a host with an Intel Core i7 processor and 16GB RAM. The network environment simulated a local area network with negligible latency (<1ms base RTT).

The load profile used for testing involved JSON payloads of approximately 2KB. This payload size was specifically calibrated to match the serialization overhead of the 30-feature input vector (V1-V28, Time, Amount) defined in the benchmark dataset (Androulaki et al., 2018). By aligning the payload size with the actual data schema required by the model, we ensured that the network throughput and serialization costs reflected a realistic production inference scenario. The ML model deployed was a Random Forest classifier pre-trained on this dataset. To evaluate performance comprehensively, we measured throughput (requests per second) and latency percentiles (p50, p90, and p99). Reporting these percentiles is crucial for assessing Service Level Objectives (SLOs), as they capture tail latency behavior that averages often hide. The recorded error rate remained below 0.1% for the AECC proposal across all tests until the point of infrastructure saturation.

4.3 Performance under Concurrent Load

The first test evaluated response latency and throughput (RPS) as the number of concurrent users simulated by Locust increased. Figure 6 presents the latency distribution, including the median (p50), p90, and p99 percentiles.

As shown, static balancing (Round Robin) distributes the load without considering worker saturation, causing latency across all percentiles to degrade rapidly as waiting queues accumulate. Critically, the p99 value (tail latency) of the static scheme increases sharply, indicating severe violations of the Service Level Objective (SLO) for 1% of users. In contrast, the AECC architecture, by consulting the control table and selecting the least-loaded worker, maintains all percentiles at low and stable levels throughout the test. This result directly addresses the challenge of latency in complex inference scenarios (Walker, 2023). It is important to note that the experimental workload was modeled using stochastic service times, randomly distributed between 100 ms and 500 ms, in order to mimic realistic inference variability caused by factors such as input vector sparsity or Python garbage collection cycles.

Under these conditions, the Static Round Robin method fails because it assumes uniform service rates. If a worker receives a batch of ”heavy” requests (approaching 500ms) by chance, Round Robin continues to send traffic to it, causing the queue to grow exponentially (as seen in the p99 tail latency spikes). The Process Control Table, however, adapts to this stochastic behavior: if a worker is bogged down by a heavy request, its counter (Ci) remains high, and the minimization function $w*$ automatically diverts new traffic to faster workers, effectively smoothing out the variance in service times.

Furthermore, the throughput analysis Figure 7 confirms that the AECC proposal achieves a maximum Requests Per Second (RPS) consistently higher than the static baseline before saturation begins, demonstrating a significant increase in operational capacity.

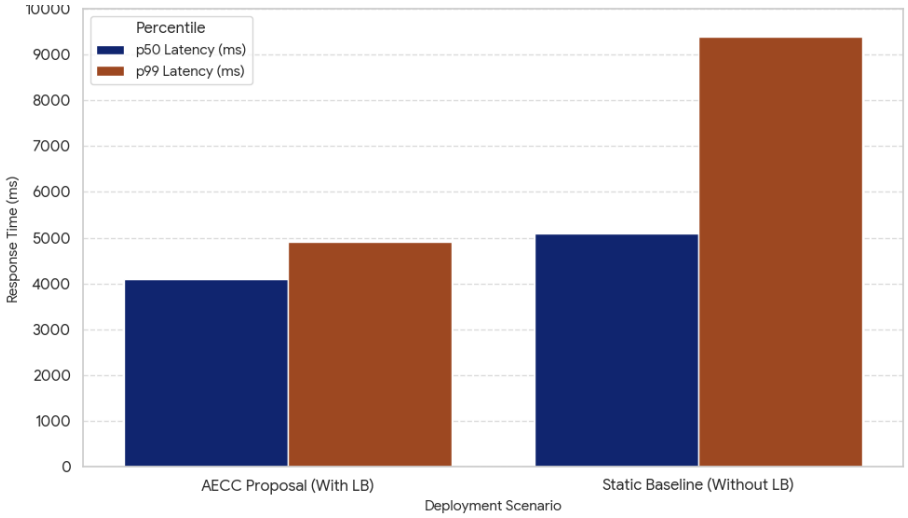

Fig. 6. Comparison of response time percentiles (p50, p99) as concurrent users increase. The AECC proposal maintains stable latency across all percentiles, validating SLO compliance.

4.4 Effectiveness of Load Distribution

The second test consisted of sending a batch of 10,000 requests to a system with 4 active workers. Fig. 8 shows the final load distribution.

The results show a very balanced distribution. This confirms that the least connections algorithm, enabled by the control table, is highly effective in preventing the formation of bottlenecks on specific workers.

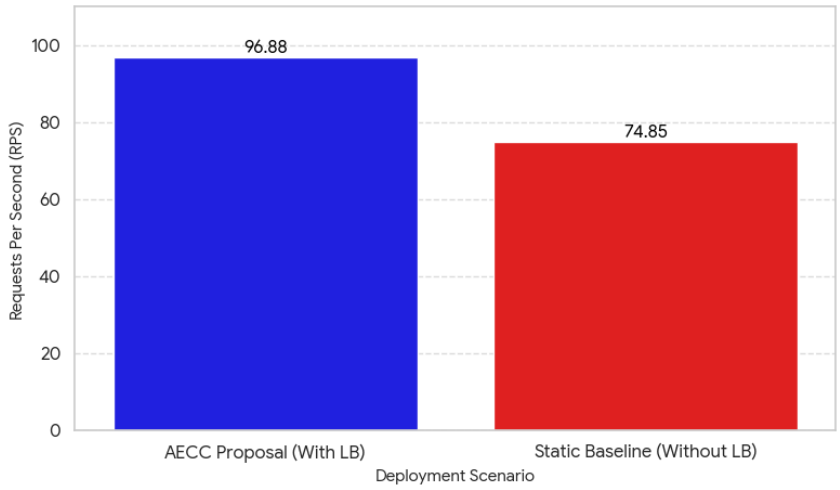

Fig.7. Comparison of maximum Throughput (RPS) achieved by the AECC proposal versus the static baseline under concurrent load.

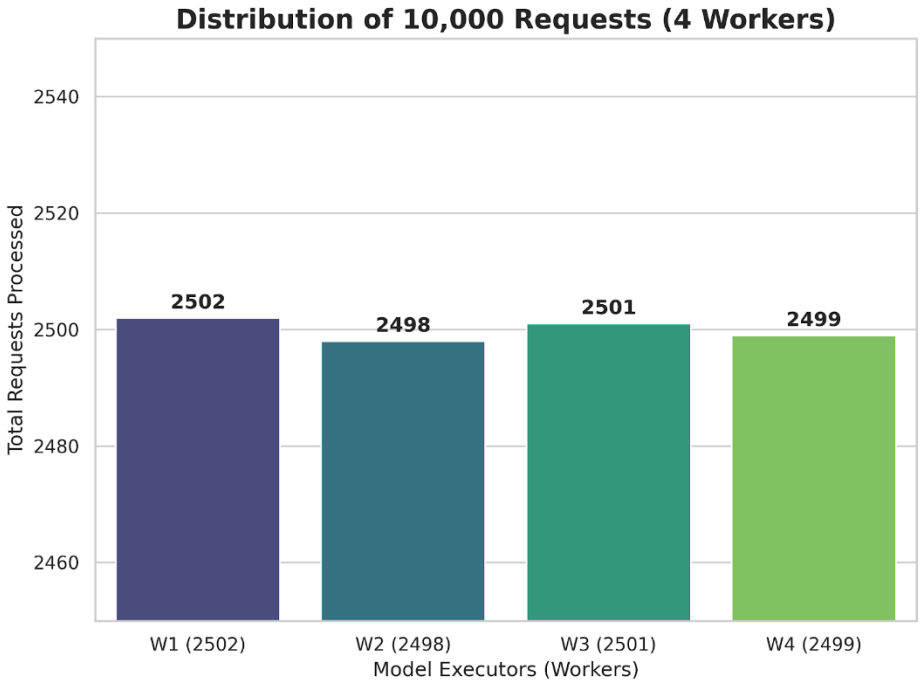

Fig. 8. Distribution of 10,000 requests among 4 active workers using the control table. A nearly equitable distribution is observed.

4.5 Validation of Monitoring and Fault Tolerance

Finally, the auto-correction logic was validated. During a load test, a worker failure was simulated (by stopping its Docker container) to test the system's Mean Time to Recovery (MTTR). The visualization dashboard (Fig. 9) reflected the system's status in real time. The Monitoring Module detected the inactivity of the failed worker and, after the configured timeout, updated its status to ERROR in the Control Table.

Immediately, the Load Manager stopped including that worker in its list of available destinations, redistributing the traffic among the remaining healthy workers. This demonstrates automatic fault tolerance, similar to the rapid recovery systems described by Kamila et al. (Kamila et al., 2022).

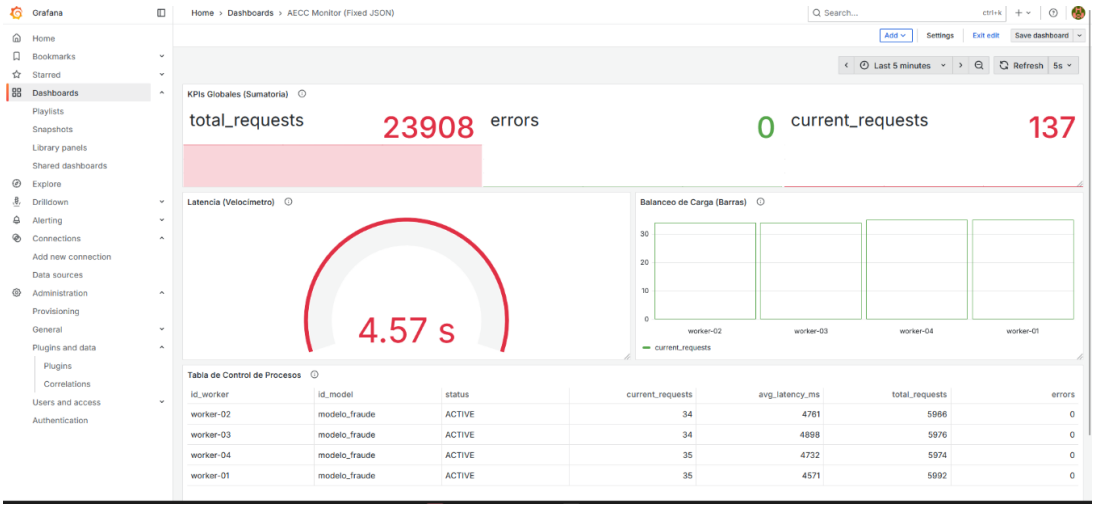

Fig. 9. Screenshot of the Grafana monitoring dashboard, showing the status, load, and mean latency of active models, and detecting a worker in an ERROR state.

Although the proposed AECC architecture showed favorable results in terms of latency stability, throughput, balanced request distribution, and automatic fault tolerance, the findings should be interpreted within the scope of the experimental setup. The evaluation was conducted in a controlled Docker-based testbed with only four worker nodes, a local-area network with negligible baseline latency, and a single pre-trained Random Forest model using approximately 2 KB JSON payloads. While this configuration was appropriate for validating the functional behavior of the architecture, it does not fully represent more heterogeneous production environments, larger-scale deployments, or workloads based on computationally heavier models.

In addition, the fault-tolerance analysis was limited to a bounded failure scenario based on stopping a worker container and verifying the system response through the Monitoring Module and the Control Table. Although this confirmed the correctness of the watchdog mechanism and the redistribution of traffic among healthy workers, the current study did not evaluate more demanding conditions such as network partitioning, Redis disruption, multiple simultaneous failures, or comparisons against industry-standard load balancers beyond the static Round Robin baseline. Therefore, the present results should be understood as a proof of architectural feasibility rather than as a definitive validation under all operational conditions.

5 Conclusions

This study presented and implemented a modular architecture centered on a process control table for managing the real-time execution of machine learning models. The results indicate that, under the evaluated scenarios, the proposed approach effectively improved the handling of concurrent requests, reduced bottlenecks, and promoted a more balanced distribution of the computational load across the available workers. In addition, the architecture supported service continuity during worker failure and enabled zero-downtime model updates, demonstrating that a centralized control mechanism can simultaneously coordinate load balancing, fault tolerance, and operational monitoring in an MLOps-oriented deployment.

Beyond execution performance, the main contribution of the study lies in the integration of orchestration and observability within a single control layer. By using the process control table as the central coordination mechanism, the system was able to track worker status, route requests dynamically, and support real-time supervision of the inference service. These results show that the proposed architecture provides a practical foundation for managing ML services in dynamic environments and offers a monitoring perspective conceptually aligned with operational supervision strategies used in IIoT-inspired systems.

Nevertheless, the results should be interpreted in light of certain limitations. The experimental evaluation was conducted in a controlled environment, using a limited number of workers and a specific deployment configuration, which may restrict the generalizability of the findings to larger-scale or more heterogeneous production settings. Furthermore, although the architecture showed promising behavior under failure scenarios and monitoring tasks, additional validation is still required to assess its performance under more demanding operational conditions and alternative balancing strategies.

Future work will focus on extending the empirical evaluation by benchmarking the proposed architecture against industry-standard load balancers such as NGINX and Traefik configured in least-connections, weighted, and predictive modes. Additional studies will also quantify the operational overhead and resource consumption of the control table components and will include systematic fault-injection experiments, such as network partition scenarios, to measure recovery metrics such as Mean Time To Recovery (MTTR). To support transparency and adoption, the reproducibility package, including the Infrastructure as Code (IaC) configuration and test scripts, is publicly available at https://github.com/Armandogma/aecc-mlops-reproducibility.

References

Aliev, K., & Antonelli, D. (2021). Proposal of a monitoring system for collaborative robots to predict outages and to assess reliability factors exploiting machine learning. *Applied Sciences, 11*(4), Article 1621. https://doi.org/10.3390/app11041621

Dal Pozzolo, A., Caelen, O., Johnson, R. A., & Bontempi, G. (2015). Calibrating probability with undersampling for unbalanced classification. In *2015 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 159–166). IEEE. https://doi.org/10.1109/SSCI.2015.33

de la Rúa Martínez, J. (2020). *Scalable architecture for automating machine learning model monitoring* [Master's thesis, KTH Royal Institute of Technology]. DiVA. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-280345

Elgamal, Z. S., El Fangary, L., & Fahmy, H. (2025). The impact of using MLOps and DevOps on container-based applications: A survey. *Informatics Bulletin, 7*(1), 51–63.

Ibadov, N., Akgün, F. M., Üncü, İ. S., Davraz, M., & Koru, M. (2025). Real-time service life estimation of vacuum insulated panels via embedded sensing and machine learning models. *Buildings, 15*(16), Article 2879. https://doi.org/10.3390/buildings15162879

Jin, Y., & Yang, Z. (2025). *Scalability optimization in cloud-based AI inference services: Strategies for real-time load balancing and automated scaling* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2504.15296

Kamila, N. K., Frnda, J., Pani, S. K., Das, R., Islam, S. M. N., Bharti, P. K., & Muduli, K. (2022). Machine learning model design for high performance cloud computing & load balancing resiliency: An innovative approach. *Journal of King Saud University–Computer and Information Sciences, 34*(10), 9991–10009. https://doi.org/10.1016/j.jksuci.2022.10.001

Low, W. K., Ramasamy, R. K., & Rajendran, V. (2024). Adaptive load balancing strategies in service composition for improved system performance. *Journal of Infrastructure, Policy and Development, 8*(13), Article 8967. https://doi.org/10.24294/jipd8967

Maddireddy, K., Methukula, S. K., Sridhar, C., & Vaidhyanathan, K. (2025). *LoCoML: A framework for real-world ML inference pipelines* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2501.14165

Mahdizadeh, M., Montazerolghaem, A., & Jamshidi, K. (2025). Task scheduling and load balancing in SDN-based cloud computing: A review of relevant research. *Journal of Engineering Research, 13*(4), 3132–3146. https://doi.org/10.1016/j.jer.2024.11.002

McClure, S., Cohen, E., Shpiner, A., Silberstein, M., Ratnasamy, S., Shenker, S., & Keslassy, I. (2025). *Load balancing for AI training workloads* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2507.21372

Moens, P., Bracke, V., Soete, C., Vanden Hautte, S., Nieves Avendano, D., Ooijevaar, T., Devos, S., Volckaert, B., & Van Hoecke, S. (2020). Scalable fleet monitoring and visualization for smart machine maintenance and industrial IoT applications. *Sensors, 20*(15), Article 4308. https://doi.org/10.3390/s20154308

Mora, S. (2024). *Intrusion detection using machine learning with real-time dashboard* [Master's thesis, National College of Ireland]. NORMA@NCI. https://norma.ncirl.ie/8752/

Puranik, S., & Mahmood, Y. (2025). *Real-time machine performance visualisation & monitoring in smart manufacturing: A dual-tool approach* [Master's thesis, Mälardalen University]. DiVA. https://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-71784

Sanjalawe, Y., Fraihat, S., Al-E'mari, S., Abualhaj, M., Makhadmeh, S., & Alzubi, E. (2025). Smart load balancing in cloud computing: Integrating feature selection with advanced deep learning models. *PLOS ONE, 20*(9), Article e0329765. https://doi.org/10.1371/journal.pone.0329765

SAP. (n.d.). *What is the industrial Internet of Things (IIoT)?* Retrieved July 30, 2026, from https://www.sap.com/resources/what-is-iiot

Shafiq, D. A., Jhanjhi, N. Z., & Abdullah, A. (2022). Load balancing techniques in cloud computing environment: A review. *Journal of King Saud University–Computer and Information Sciences, 34*(7), 3910–3933. https://doi.org/10.1016/j.jksuci.2021.02.007

Sliwko, L. (2024). Cluster workload allocation: A predictive approach leveraging machine learning efficiency. *IEEE Access, 12*, 194091–194107. https://doi.org/10.1109/ACCESS.2024.3520422

Vashistha, D., Mehta, D., Kumhar, M., Bhatia, J., & Alkhayyat, A. (2025). Deep learning based load balancing in cloud computing: A survey. *Procedia Computer Science, 259*, 1963–1972. https://doi.org/10.1016/j.procs.2025.04.152

Walker, E. (2023). Challenges and solutions in deploying machine learning models at scale. *American Journal of Machine Learning, 4*(5), 13–22.

Yalamati, S. (2025). AI-enhanced fault tolerance in microservices: Predictive failure models for resilient software systems. *International Journal of Applied Engineering & Technology, 7*(2), 26–35.

Yao, J., Zhang, L., & Huang, J. (2025). *Evaluation of large language model-driven AutoML in data and model management from human-centered perspective* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2507.05962

Yao, Z., Desmouceaux, Y., Townsley, M., & Heide Clausen, T. (2021). *Towards intelligent load balancing in data centers* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2110.15788

Yu, Y., Abadi, M., Barham, P., Brevdo, E., Burrows, M., Davis, A., Dean, J., Ghemawat, S., Harley, T., Hawkins, P., Isard, M., Kudlur, M., Monga, R., Murray, D., & Zheng, X. (2018). Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference* (pp. 1–15). Association for Computing Machinery. https://doi.org/10.1145/3190508.3190551

Zhang, Z., Wu, Z., Rincon, D., & Christofides, P. D. (2019). Real-time optimization and control of nonlinear processes using machine learning. *Mathematics, 7*(10), Article 890. https://doi.org/10.3390/math7100890