Dynamic Load Balancing and Fault Tolerance in ML Deployments Using a Process Control Table

Guadarrama-Osorio, Luis-Armando*, Cruz-Mendoza, Mariana-Carolyn *, Sánchez-DelaCruz, Eddy†, and Primero-Huerta, Cesar *

*División de Ingeniería en Sistemas Computacionales, Tecnológico Nacional de México-Tecnológico de Estudios Superiores de Valle de Bravo, km. 30 de la Carretera Monumento-Valle de Bravo 51200, Mexico

†Artificial Intelligence Lab., Tecnológico Nacional de México-Instituto Tecnológico Superior de Misantla, Misantla, 93828 , Veracruz, México.

Corresponding author: Primero-Huerta, César. iscprimerocesar@gmail.com

***Abstract*—Putting Machine Learning (ML) models into production can be a significant challenge, especially when managing a high volume of requests and monitoring real-time performance. Many current solutions address these problems in isolation, which can create bottlenecks and increase latency. This paper presents the design of a Process Control Table, which functions as a central coordinator to orchestrate the execution of ML models in real time. The table allows for dynamic traffic routing, assigning each new task to the least occupied *worker* (executor) at that moment. Test results with a prototype indicate that this approach not only reduces latency but also keeps it stable even under high demand, unlike traditional schemes that tend to saturate. Furthermore, the table promotes equitable load balancing and features automatic failover mechanisms, which increases the reliability of model deployments.**

***Index Terms*—MLOps, Load Balancing, Model Monitoring, Request Optimization, Technological Integration, Task Management, Process Management.**

I. Introduction

The use of Machine Learning (ML) has grown rapidly beyond the academic sphere, becoming a key element of business applications. This expansion can be seen in a variety of fields, from real-time optimization of chemical processes [1] to predictive monitoring in smart manufacturing [2, 3], and even in enhancing cybersecurity through intrusion detection [4]. However, a significant barrier still exists between the development of a functional model in the lab and its successful implementation in production environments [5]. Overcoming this barrier is crucial, as the true value of ML is revealed not just in experimentation, but primarily in its reliable and scalable operation in production [5].

There are two main challenges in implementing ML models, a process known as MLOps (ML Operations) [6]. The first is managing real-time execution: models in production must be able to handle volumes of inference requests that are unpredictable and variable. Without an adequate management system, concurrency can overwhelm server resources, creating bottlenecks that result in high latency and potential failures [5, 7]. The second challenge is managing the model lifecycle. A deployed model, which is available for consumption by external applications, is not static; it needs to be continuously monitored to detect any degradation in its performance, a phenomenon known as model drift [6, 8]. Conventional monitoring systems, often based on batch processing, tend to operate with considerable delays, often hours, which is not sufficient for applications requiring real-time responses or detections [8].

A review of the current literature, detailed in Section II, shows that existing research tends to address ML deployment challenges in a fragmented manner. On one hand, there are studies focused on optimizing load distribution in cloud environments [9, 10] and improving reliability or fault recovery capabilities [7]. On the other hand, we find works that center on MLOps-specific monitoring to detect model performance degradation (drift) [8] or on the real-time visualization of key performance indicators (KPIs), often from an industrial perspective [2, 11]. However, a significant limitation is highlighted: the lack of an integrated architectural design that combines three essential functions: dynamic load distribution, real-time status monitoring, and model lifecycle management, all under a centralized and efficient control mechanism.

To overcome this obstacle, this project proposes the design and implementation of a *process control table*. The proposal is based on concurrency control principles, which is a key challenge in large-scale computing [5]. Efficiently managing simultaneous requests is fundamental for load balancing and is crucial for fault tolerance and high performance [7, 9].

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

The methodology used in this research was experimental. We carried out the design and implementation of a functional prototype that integrates key components, such as the Control Module and the workers (executors) in containers. To ensure the results are applicable to critical production environments, the experimental design replicates a high-frequency financial fraud detection scenario, a standard benchmark for validating latency-sensitive MLOps architectures. This prototype was evaluated through load tests (simulations) to measure and validate its performance in high-demand situations.

The work is organized into several sections. Section II explores related works and the state of the art. Section III details the design methodology, the proposed system architecture, and the structure of the control table. In Section IV, the performance test results are presented and discussed. Finally, Section V summarizes the conclusions and outlines future work.

II. Related Work

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 [6, 8, 12]. Below, we review previous research in these areas, which serves as the foundation for this research.

A. MLOps and Deployment Challenges

Migrating an ML model from a laboratory environment to a large-scale production system represents a challenge for organizations, as it involves complex infrastructure, inference performance, and data management challenges [5]. The practice of *MLOps* seeks to apply *DevOps*[1] principles to automate and standardize the entire ML lifecycle [6], An approach that is particularly relevant for modern container-based applications is mentioned in [13]. The main challenges when deploying at scale include the need for infrastructure that adapts to demand, managing data processing bottlenecks, optimizing inference latency, and automating repetitive tasks [5, 14]. To succeed in MLOps, lifecycle automation and efficient load balancing are considered fundamental [5, 12].

In addition to the initial implementation, a key aspect of *MLOps* is continuous monitoring to identify *model drift*, which refers to the performance decrease when real-world data changes. This monitoring is essential to ensure the model's reliability and performance over time, and it becomes a fundamental pillar in data and model management on modern cloud platforms [6, 8, 15].

B. Load Balancing for Inference

Managing the execution of machine learning models in real time is closely linked to the system's ability to handle unpredictable and variable volumes of inference requests [12]. Load balancing is responsible for distributing these requests among multiple servers or *workers*[2] in order to prevent overload and reduce latency [9]. Generally, balancing strategies are classified as static (such as *round robin*) and dynamic (which are based on the current load). Dynamic methods, which make decisions based on the system's state, are considered more effective for handling variable workloads in the cloud [9, 16].

To overcome the limitations of dynamic methods (which only react to the instantaneous state), the current trend is oriented towards smarter balancing methods, capable of predicting and actively managing load allocation [17, 18]. This includes the use of *deep learning* techniques to model and anticipate system loads, optimizing task assignment [10, 19]. Complementarily, centralized traffic management using software-defined networking (SDN) allows for more flexible resource orchestration [20]. These balancing principles are fundamental for both the high performance of inference workloads [12] and AI training workloads [21].

C. Real-time Monitoring and Predictive Maintenance

A significant challenge in traditional MLOps systems is monitoring latency. Many of them operate in batches, which means that model drift detection can take hours. For critical applications, real-time monitoring is essential. In this context, [8] proposed a *streaming* architecture (Spark and Kafka), which can detect drift in a matter of seconds. This highlights the need for immediate processing for monitoring and visualization of key performance indicators (KPIs) [2, 11].

[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) [6, 13].

[2]A service instance designed to receive units of work.

This concept of immediate KPI visualization is also fundamental in industry (IIoT[3]). For example, [2] implemented a real-time monitoring system using Grafana for instant operation and Power BI for historical analysis. Similarly, [11] designed a three-tier architecture to monitor fleets of machines and generate data for predictive maintenance.

D. Proactive Systems and Fault Tolerance

Real-time monitoring forms the basis for designing proactive systems capable of anticipating failures. Various studies apply ML to predict problems from sensor data. [3] used ML to analyze the wear of collaborative robots (cobots) and predict stops. In a similar vein, [23] used *Random Forest* models with integrated sensor data to estimate the remaining useful life of components with high accuracy.

This predictive approach also applies to cloud infrastructure. To address failures reactively, [7] showed that using ML (e.g., XGBoost) on performance metrics (CPU, memory) allows for predicting service outages and speeding up recovery by 38.15 %. Employing AI to build predictive failure models is a central strategy for high-reliability microservices systems [24]. The feasibility of these systems also depends on software-level optimizations, such as process improvement using neural networks [1], implementing efficient control primitives in ML *frameworks* [25], and their integration into security *dashboards* [4].

From the analysis of these works, a central limitation emerges: the literature offers effective solutions for *MLOps* [5, 6, 13], load balancing [9, 12, 19], and real-time monitoring [2, 7, 8], but almost always as independent systems. There is a lack of architectures that use a single, centralized, low-latency data structure to simultaneously coordinate these three tasks: managing dynamic load balancing, facilitating real-time monitoring, and administering the model lifecycle (updates or self-correction).

This research addresses this gap through the design of a **Process Control Table** that coordinates the three aforementioned tasks. The table acts as a centralized state and single point of coordination for the execution [14, 25], load distribution [9, 12], and monitoring [2, 8] of ML models in real time.

III. Methodology

For this research, a practical approach was used that focused on the development and experimental evaluation of a prototype. The process was organized into five main phases, which were addressed sequentially. .

A. Phase 1: Architecture Design (AECC)

In the first phase, we defined a design that we call the Control-Centered ML Execution Architecture (AECC). This design is based on a modular approach, inspired by three-tier architectures used for monitoring machine fleets [11], and organizes the system into four main components (see Fig. 2). This division of responsibilities makes system management and maintenance much simpler. The components are detailed below.

1) 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.

[3]Industrial Internet of Things (IIoT) is the application of the Internet of Things (IoT) to the industrial sector [22].

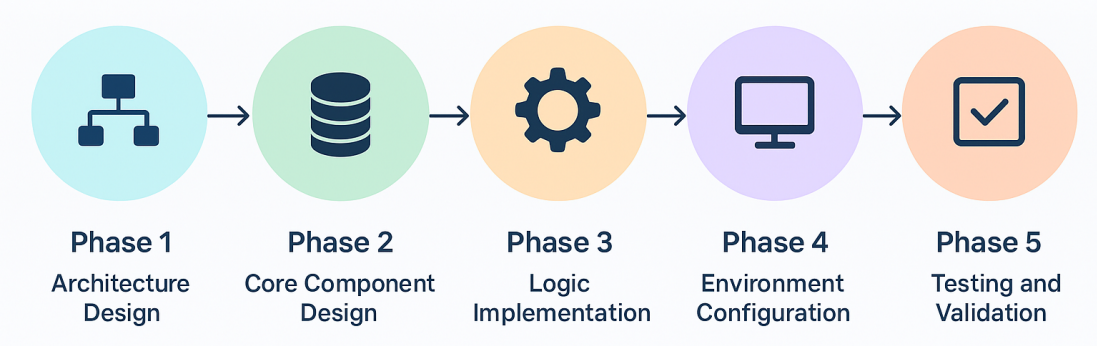

Figure 1. Methodology used.

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.

2) 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.
3) 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.
4) 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 [2], 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 [7].

B. Phase 2: Process Control Table Design

As the central component, its design was a necessary phase. A key-value data structure optimized for high-speed operations was defined. It was determined that this table should be implemented in an in-memory database using Redis to ensure minimal latency. Regarding overhead, the system utilizes pipelined operations to keep the read/write latency impact below 0.5 ms per request, ensuring that the control mechanism does not become a bottleneck. To address resilience, although the current prototype utilizes a standalone Redis instance to isolate algorithmic performance, the architectural design explicitly decouples the logic from the storage topology. The Control Module's connection interface is implemented using standard drivers compatible with Redis Sentinel and Cluster modes. This design choice ensures that migrating to a High Availability (HA) configuration requires only configuration updates rather than code refactoring, effectively eliminating the Single Point of Failure (SPOF) risk inherent in centralized control structures.

Furthermore, a heartbeat mechanism was implemented via the last_activity field acting 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 defined threshold, the Monitoring Module automatically marks the worker as inactive, effectively handling node failures without manual intervention.

The table structure (see Table I) was designed to store the complete, real-time status of every active worker in the system. Each field was selected to serve a specific purpose in the control and monitoring logic.

Table I
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) [11].
`last_activity`	(Timestamp)	To detect inactive workers.

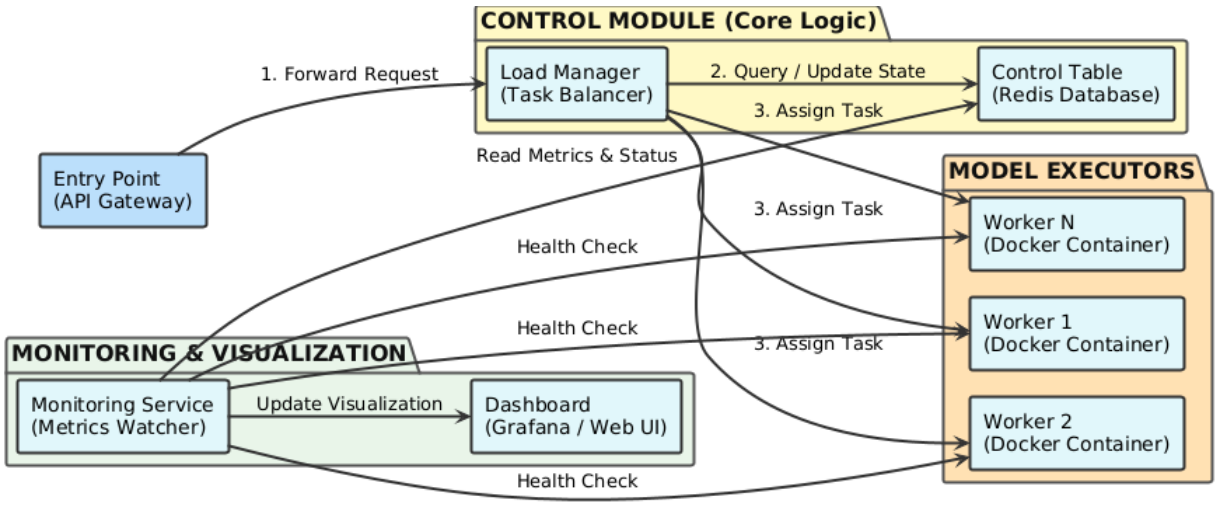

Figure 2. AECC architecture diagram. Shows the flow of a request from the API Gateway, through the Control Module (which queries the Control Table) and being directed to one of the Model Executors. The Monitoring Module reads the table for the dashboard.

C. 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 [9]. 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 I 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 [7].

- Monitoring (Read): A process (Grafana [2]) 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 ($\mathbf{p}$) of the two distributions being the same falls below a strict limit ($\mathbf{p < 0.05}$), 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.

D. 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.

E. 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.

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

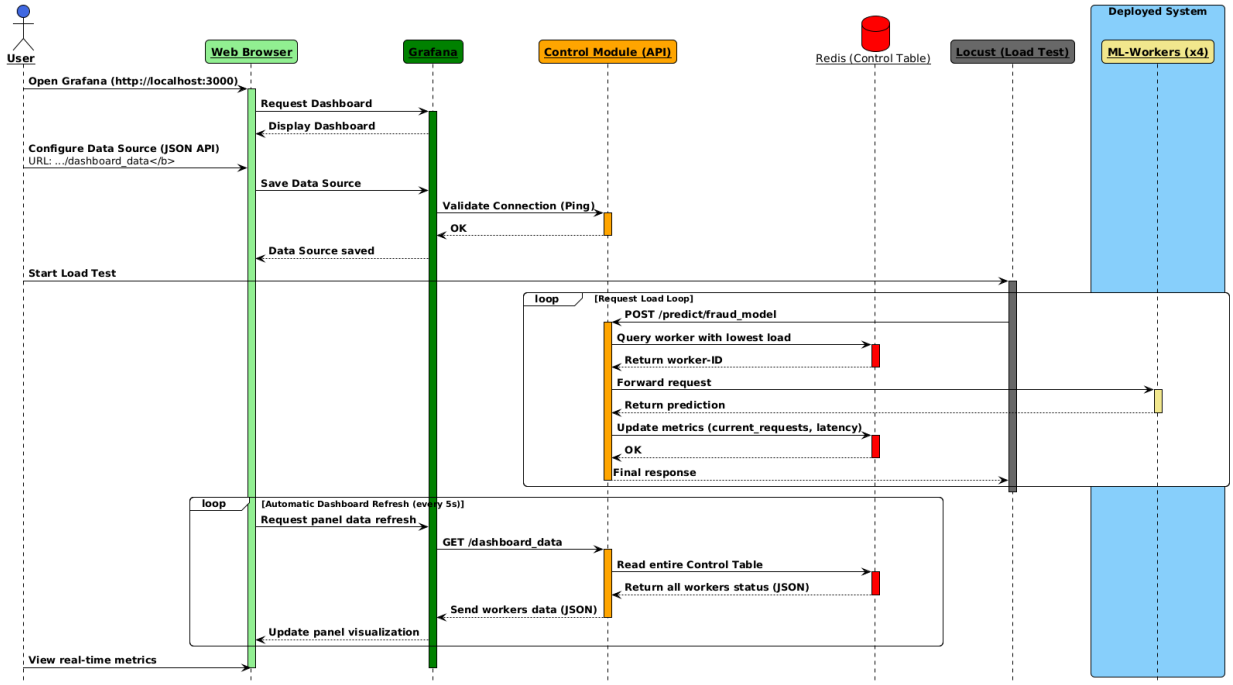

Figure 3. Sequence diagram of the real-time monitoring and prediction flow.

2) 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.

IV. Results and Discussion

The prototype described in section III-D was subjected to the defined load tests. The results validate the utility of the AECC architecture compared to the static approach.

A. 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 [26]. 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.

B. Performance under Concurrent Load

The first test measured the response latency and throughput (RPS) while increasing the number of concurrent users simulated by Locust. Figure 4 presents the latency distribution, showing the median (p50), p90, and p99 percentiles.

As observed, static balancing (Round Robin) distributes the load without considering worker saturation, causing the latency for all percentiles to degrade rapidly as waiting queues form. Critically, the p99 (tail latency) for the static scheme rises sharply, indicating severe Service Level Objective (SLO) violation for 1% of users. In contrast, the AECC architecture, by consulting the control table and choosing the worker with the least load, maintains all percentiles low and stable throughout the test duration. This result addresses the challenge of latency in complex inferences [5].

Furthermore, the throughput analysis Figure 5 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.

C. Effectiveness of Load Distribution

The second test consisted of sending a batch of 10,000 requests to a system with 4 active workers. Fig. 6 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.

D. 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

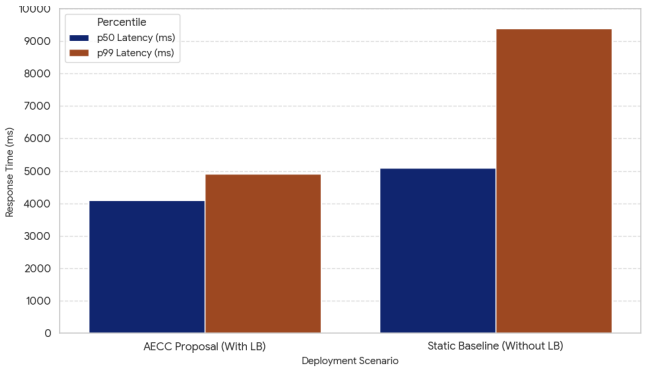

Figure 4. Comparison of response time percentiles (p50, p90, p99) as concurrent users increase. The AECC proposal maintains stable latency across all percentiles, validating SLO compliance.

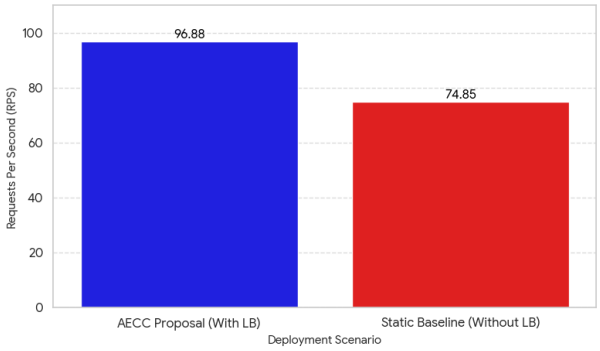

Figure 5. Comparison of maximum Throughput (RPS) achieved by the AECC proposal versus the static baseline under concurrent load.

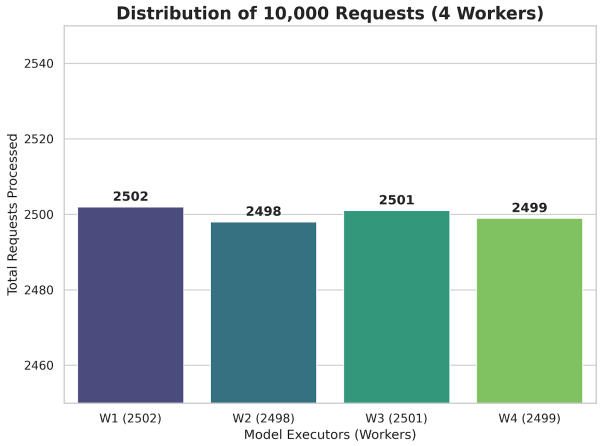

Figure 6. Distribution of 10,000 requests among 4 active workers using the control table. A nearly equitable distribution is observed.

test the system's Mean Time to Recovery (MTTR). The visualization dashboard (Fig. 7) 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. [7].

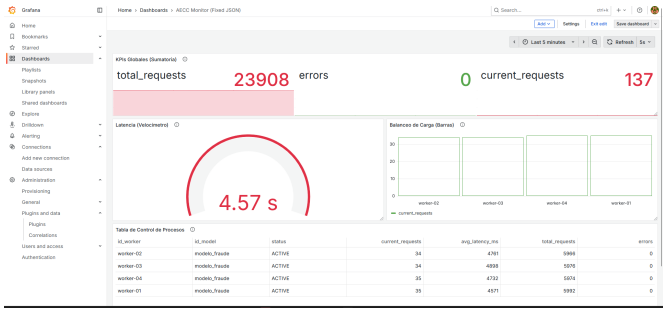

Figure 7. Screenshot of the monitoring dashboard, showing the status, load, and mean latency of active models, and detecting a worker in an ERROR state.

V. Conclusion

In this work, a modular architecture centered on a process control table was designed and implemented, with the objective of managing the execution of ML models in real time. The results show that this approach effectively addresses problems related to concurrent requests and bottlenecks, thanks to dynamic load balancing.

The proposed architecture not only optimizes execution performance, but also facilitates the MLOps lifecycle, allowing for zero-downtime model updates and offering a solid foundation for real-time monitoring, similar to fleet monitoring systems in IIoT [2, 11].

Future work will focus on deepening the empirical evaluation based on the limitations identified. Specifically, we aim to benchmark the proposed architecture against industry-standard load balancers like NGINX and Traefik configured in "least connections" mode, as well as weighted and predictive schemes. We also plan to quantify the exact operational overhead and resource consumption of the control table components. Furthermore, we intend to perform systematic fault injection tests (e.g., network partitioning) to measure Mean Time To Recovery (MTTR) precisely.Finally, to facilitate adoption and traceability, the complete reproducibility package, including the Infrastructure as Code (IaC) configuration and test scripts, has been released and is publicly available at https://github.com/Armandogma/aecc-mlops-reproducibility.

References

[1] Z. Zhang, Z. Wu, D. Rincon, and P. D. Christofides, "Real-time optimization and control of nonlinear processes using machine learning," *Mathematics*, vol. 7, no. 10, p. 890, 2019.

[2] S. Puranik and Y. Mahmood, "Real-time machine performance visualisation & monitoring in smart manufacturing: A dual-tool approach," Master's thesis, Mälardalens universitet, 2025.

[3] K. Aliev and D. Antonelli, "Proposal of a monitoring system for collaborative robots to predict outages and to assess reliability factors exploiting machine learning," *Applied Sciences*, vol. 11, no. 4, p. 1621, 2021.

[4] S. Mora, "Intrusion detection using machine learning with real-time dashboard," 2024, mSc Research Project.

[5] E. Walker, "Challenges and solutions in deploying machine learning models at scale," *American Journal of Machine Learning*, vol. 4, no. 5, pp. 13–22, 2023.

[6] L. Emma, "Mlops on cloud platforms: End-to-end ci/cd pipelines for machine learning model deployment and monitoring," March 2025, preprint.

[7] N. K. Kamila, J. Frnda, S. K. Pani, R. Das, S. M. Islam, P. Bharti, and K. Muduli, "Machine learning model design for high performance cloud computing & load balancing resiliency: An innovative approach," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 9991–10 009, 2022.

[8] J. de la Rúa Martínez, "Scalable architecture for automating machine learning model monitoring," Master's thesis, KTH Royal Institute of Technology, 2020.

[9] D. A. Shafiq, N. Jhanjhi, and A. Abdullah, "Load balancing techniques in cloud computing environment: A review," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 8, pp. 3910–3933, 2022.

[10] D. Vashistha, D. Mehtaa, M. Kumharb, J. Bhatiab, and A. Alkhayyat, "Deep learning based load balancing in cloud computing: A survey," *Procedia Computer Science*, vol. 259, pp. 1963–1972, 2025.

[11] P. Moens, V. Bracke, C. Soete, S. Vanden Hautte, D. Nieves Avendano, T. Ooijevaar, S. Devos, B. Volckaert, and S. Van Hoecke, "Scalable fleet monitoring and visualization for smart machine maintenance and industrial iot applications," *Sensors*, vol. 20, no. 15, p. 4308, 2020.

[12] R. Gupta and P. Sharma, "Scalability optimization in cloud-based ai inference services: Strategies for real-time load balancing and automated scaling," in *2024 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. IEEE, 2024, pp. 320–327.

[13] E. A. Al-Hawari and T. S. Jaber, "The impact of using mlops and devops on container-based applications: A survey," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 15, no. 3, pp. 1–9, 2024.

[14] J. Chen, W. Li, and J. Zhang, "Locoml: A framework for real-world ml inference pipelines," in *Proceedings of the 2024 USENIX Conference on Operationalizing Machine Learning (OpML '24)*. USENIX Association, 2024, pp. 78–89.

[15] L. Wang, W. Chen, and X. Ma, "Evaluation of large language model driven automl in data and model management from human-centered perspective," *Journal of Data and Model Management*, vol. 8, no. 1, pp. 45–59, 2025.

[16] J. Li and Z. Wu, "Adaptive load balancing strategies in service composition for improved system performance," *Journal of Cloud Computing*, vol. 13, no. 1, pp. 1–18, 2024.

[17] D. Al-Jumeily, M. M, and O. Al-Jumeily, "Towards intelligent load balancing in data centers," in *2024 17th International Conference on Developments in eSystems Engineering (DeSE)*. IEEE, 2024, pp. 245–251.

[18] J. Adebayo and L. O'Connor, "Cluster workload allocation: A predictive approach leveraging machine learning," in *2024 IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2024, pp. 112–119.

[19] W. Zhang and Y. Li, "Smart load balancing in cloud computing: Integrating feature selection with advanced deep learning models," *Future Generation Computer Systems*, vol. 163, pp. 135–148, 2025.

[20] M. Mahdizadeh, A. Montazerolghaem, and K. Jamshidi, "Task scheduling and load balancing in sdn-based cloud computing: A review of relevant research," *Journal of Engineering Research*, 2024.

[21] J. Boutros and R. K, "Load balancing for ai training workload," *Journal of Grid Computing*, vol. 23, no. 1, pp. 1–22, 2025.

[22] SAP, "What is the industrial internet of things (iiot)?" https://www.sap.com/latinamerica/products/scm/industry-4-0/what-is-iiot.html, 2025, accedido: 24 de octubre de 2025.

[23] N. Ibadov, F. M. Akgün, İ. S. Üncü, M. Davraz, and M. Koru, "Real-time service life estimation of vacuum insulated panels via embedded sensing and machine learning models," *Buildings*, vol. 15, no. 16, p. 2879, 2025.

[24] M. Kaur and R. Singh, "Ai-enhanced fault tolerance in microservices: Predictive failure models for resilient software systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 35, no. 2, pp. 211–230, 2025.

[25] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins *et al.*, "Dynamic control flow in large-scale machine learning," in *13th {EuroSys} Conference 2018 (EuroSys '18)*, 2018, pp. 1–15.

[26] A. Dal Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi, "Calibrating probability with undersampling for unbalanced classification," in *2015 IEEE Symposium Series on Computational Intelligence*. IEEE, 2015, pp. 159–166.