**International Journal of Computer Engineering and Technology (IJCET)**
Volume 16, Issue 2, March-April 2025, pp. 160-178, Article ID: IJCET_16_02_011
Available online at https://iaeme.com/Home/issue/IJCET?Volume=16&Issue=2
ISSN Print: 0976-6367; ISSN Online: 0976-6375; Journal ID: 5751-5249
Impact Factor (2025): 18.59 (Based on Google Scholar Citation)
DOI: https://doi.org/10.34218/IJCET_16_02_011

# THE ROLE OF MACHINE LEARNING IN CONTAINER ORCHESTRATION: SMARTER AUTOSCALING AND MONITORING

**Ramesh Krishna Mahimalur**
CNET Global Solutions, Inc., Richardson, TX 75080 USA.

## ABSTRACT

*Container orchestration systems like Kubernetes have become the backbone of modern cloud-native applications, enabling automated deployment, scaling, and management of containerized applications. However, traditional rule-based approaches to autoscaling and monitoring face challenges in dynamic workload environments, often leading to resource inefficiencies or performance degradation. This paper explores how machine learning techniques can enhance container orchestration with smarter, more adaptive mechanisms. The research investigates reinforcement learning models for predictive autoscaling, anomaly detection for proactive monitoring, and time series forecasting for resource optimization. Experimental results demonstrate that ML-augmented orchestration systems can achieve up to 27% better resource utilization while reducing SLA violations by 18% compared to threshold-based approaches. Additionally, the implementation of ML-based anomaly detection identified 92% of performance issues before they affected user experience. The findings suggest that integrating machine learning with container orchestration provides significant advantages in managing the complexity and dynamism of modern microservices* *environments, though challenges remain in training data requirements and real-time inference capabilities.*

**Keywords:** Container Orchestration, Machine Learning, Kubernetes, Predictive Autoscaling, Anomaly Detection, Resource Optimization, Time Series Forecasting

**Cite this Article:** Ramesh Krishna Mahimalur. The Role of Machine Learning in Container Orchestration: Smarter Autoscaling and Monitoring. *International Journal of Computer Engineering and Technology (IJCET)*, 16(2), 2025, 160-178.

https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_16_ISSUE_2/IJCET_16_02_011.pdf

## I. Introduction

Container orchestration has revolutionized the way applications are deployed and managed in cloud environments. Systems like Kubernetes, Docker Swarm, and Amazon ECS automate the deployment, scaling, and operations of application containers across clusters of hosts. While these systems provide robust capabilities for managing containerized applications, they primarily rely on threshold-based rules and reactive approaches to handle scaling decisions and detect anomalies.

As cloud-native applications grow in complexity and scale, traditional rule-based orchestration faces several limitations. Static thresholds for autoscaling cannot adapt to changing application behavior or anticipate future load patterns. Similarly, conventional monitoring systems often detect issues only after they impact service quality. These limitations have motivated the exploration of more intelligent approaches to container orchestration.

Machine learning (ML) presents promising opportunities to address these challenges by enabling predictive, adaptive, and context-aware orchestration capabilities. ML algorithms can learn from historical patterns, adapt to changing conditions, and make decisions based on complex relationships that would be difficult to capture with static rules.

This paper investigates the application of machine learning techniques to enhance container orchestration systems, focusing on three key areas: predictive autoscaling, anomaly detection for monitoring, and resource optimization. The research aims to determine how ML-augmented orchestration systems compare to traditional approaches in terms of resource efficiency, performance stability, and operational reliability.

The contributions of this paper include:

1. A framework for integrating machine learning models into container orchestration systems
2. Experimental evaluation of reinforcement learning approaches for predictive autoscaling
3. Novel anomaly detection techniques for proactive monitoring of containerized applications
4. Time series forecasting methods for optimizing resource allocation
5. Practical implementation guidelines and lessons learned from real-world deployments

## II. RELATED WORK

### 2.1 Traditional Container Orchestration

Container orchestration systems have evolved significantly over the past decade. Kubernetes, originally developed by Google and now maintained by the Cloud Native Computing Foundation, has emerged as the dominant solution. Burns et al. [1] described the architecture and design principles of Kubernetes, highlighting its approach to container scheduling, service discovery, and cluster management.

Traditional autoscaling in container orchestration typically relies on threshold-based rules. Horizontal Pod Autoscaler (HPA) in Kubernetes, for example, adjusts the number of pod replicas based on CPU utilization or custom metrics [2]. While effective for predictable workloads, these approaches struggle with variable traffic patterns and complex application behaviors.

### 2.2 Machine Learning in Cloud Resource Management

Research on applying machine learning to cloud resource management has gained momentum in recent years. Pietri and Sakellariou [3] surveyed various ML techniques for resource provisioning in cloud environments, highlighting the potential of reinforcement learning and neural networks for adaptive scaling decisions.

For anomaly detection in cloud systems, Chandola et al. [4] provided a comprehensive overview of techniques, including statistical methods, clustering approaches, and deep learning models. These methods have shown promise in identifying unusual patterns that might indicate system failures or security breaches.

### 2.3 Predictive Autoscaling

Several studies have explored predictive approaches to autoscaling. Gong et al. [5] proposed a time series forecasting method using ARIMA models to predict future resource demands for VMs. More recently, deep learning approaches have been applied to this problem. Zhang et al. [6] demonstrated how recurrent neural networks (RNNs) can capture temporal dependencies in cloud workloads for more accurate scaling decisions.

Reinforcement learning (RL) has also shown promise for autoscaling decisions. Dutreilh et al. [7] applied Q-learning to VM autoscaling, while Xu et al. [8] extended this approach to containerized environments, showing improved resource efficiency compared to threshold- based methods.

### 2.4 Anomaly Detection for Monitoring

Machine learning for anomaly detection in cloud monitoring has been explored in various contexts. Wang et al. [9] used unsupervised learning to detect anomalies in cloud infrastructure metrics. Gulenko et al. [10] applied deep learning techniques for real-time anomaly detection in microservices architectures, showing significant improvements over traditional monitoring approaches.

## III. ML-ENHANCED CONTAINER ORCHESTRATION FRAMEWORK

This section presents a framework for integrating machine learning capabilities into container orchestration systems. The framework addresses three core orchestration functions: autoscaling, monitoring, and resource optimization, as illustrated in Fig. 1.

### 3.1 System Architecture

The proposed framework augments existing container orchestration systems with machine learning components that operate alongside traditional mechanisms. The ML layer consists of three main components:

1. Predictive Autoscaler: Uses reinforcement learning to make proactive scaling decisions based on historical patterns and current state
2. Anomaly Detector: Employs unsupervised learning to identify unusual system behavior before it impacts performance
3. Resource Optimizer: Leverages time series forecasting to predict resource requirements and optimize allocation Fig. 1 shows the high-level architecture of the ML-enhanced orchestration framework:

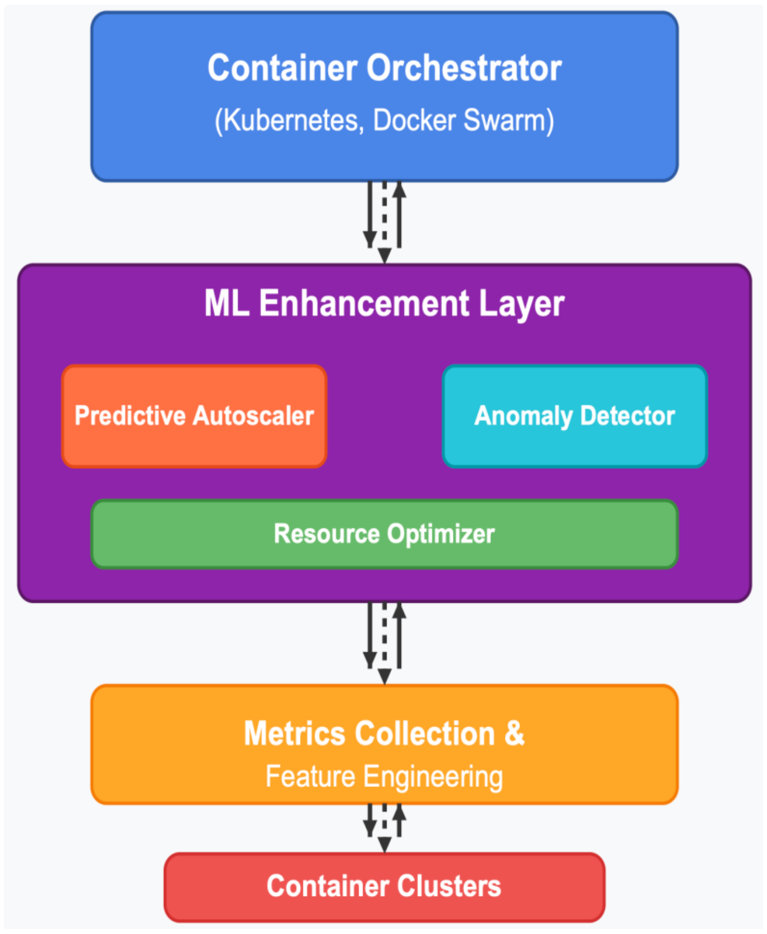

Figure 1: Architecture of ML-Enhanced Container Orchestration Framework

The ML components interact with the orchestration system through APIs and custom resource definitions. They consume metrics from the monitoring system, apply machine learning algorithms to those metrics, and influence orchestration decisions through the existing control interfaces.

### 3.2 Data Collection and Feature Engineering

Effective machine learning for container orchestration requires comprehensive data collection and feature engineering. The framework collects the following categories of metrics:

1. Resource utilization metrics (CPU, memory, network, disk)
2. Application-level metrics (request rates, latency, error rates)
3. Environment metrics (time of day, day of week, seasonal patterns)
4. Inter-service dependencies and communication patterns

These raw metrics are then transformed into features suitable for machine learning models through processes such as normalization, aggregation, and dimensionality reduction. Feature engineering is critical for capturing the temporal patterns and relationships between metrics that indicate impending issues or resource needs.

## 3.3 Predictive Autoscaling with Reinforcement Learning

The predictive autoscaling component employs reinforcement learning to make scaling decisions that optimize both resource utilization and application performance. The RL approach frames autoscaling as a Markov Decision Process (MDP) with the following components:

State: The current system state, including resource utilization, request rates, and application performance metrics  Actions: Scaling decisions (scale up, scale down, or maintain current scale)

Reward: A function that balances resource efficiency and performance objectives Policy: The strategy for selecting actions based on the current state

The state space is defined as: S = {cpu_util, mem_util, req_rate, latency, error_rate, time_features} Where time_features captures temporal patterns (hour of day, day of week).

The action space is defined as: $A = \{-n, -n+1, ..., -1, 0, 1, ..., n-1, n\}$

Where negative values represent scaling down by the specified number of instances, positive values represent scaling up, and 0 represents maintaining the current scale.

The reward function balances resource efficiency and performance:

$$R(s, a, s') = w_1 \times \text{performance\_score} - w_2 \times \text{resource\_cost}$$

Where $w_1$ and $w_2$ are weights that determine the relative importance of performance versus cost.

The RL agent uses a Deep Q-Network (DQN) to approximate the Q-function, which estimates the expected long-term reward of taking a particular action in a given state. The DQN is trained on historical data and continuously updated with new observations.

The following pseudocode describes the RL-based autoscaling algorithm:

```
def rl_autoscaler(current_state):
    # Preprocess the current state
    state_features = preprocess(current_state)

    # Use DQN to select the best action
    action = dqn.select_action(state_features)

    # Apply the selected action
    new_replicas = current_replicas + action

    # Enforce min/max constraints
    new_replicas = max(min_replicas, min(max_replicas, new_replicas))

    # Update the scaling target
    if new_replicas != current_replicas:
        apply_scaling_decision(new_replicas)

    # Observe the result and update the model
    next_state = observe_new_state()
    reward = calculate_reward(current_state, action, next_state)
    dqn.update(state_features, action, reward, next_state)

    return new_replicas

```

## 3.4 Anomaly Detection for Proactive Monitoring

The anomaly detection component uses unsupervised learning to identify unusual patterns in system behavior that may indicate impending issues. The approach combines multiple techniques to detect different types of anomalies:

1. Isolation Forest for point anomalies: Identifies individual observations that deviate significantly from normal patterns
2. Autoencoder for contextual anomalies: Detects anomalies in the relationships between different metrics
3. Time series decomposition for seasonal anomalies: Identifies deviations from expected seasonal patterns

The anomaly detection pipeline processes metrics in real-time and generates alerts when anomalies are detected. The severity of an alert is determined by the confidence score of the anomaly and its potential impact on system performance.

The following pseudocode illustrates the anomaly detection process:

```
def detect_anomalies(metrics_batch):
    # Preprocess and normalize metrics
    preprocessed_metrics = preprocess(metrics_batch)

    # Apply different detection methods
    point_anomalies = isolation_forest.predict(preprocessed_metrics)

    # Reconstruct metrics using autoencoder
    reconstructed = autoencoder.predict(preprocessed_metrics)
    reconstruction_error = mse(preprocessed_metrics, reconstructed)
    contextual_anomalies = reconstruction_error > threshold

    # Decompose time series and check for seasonal anomalies
    expected_seasonal = time_series_model.predict(time_features)
    seasonal_deviation = abs(preprocessed_metrics - expected_seasonal)
    seasonal_anomalies = seasonal_deviation > seasonal_threshold

    # Combine results with weighted voting
    combined_score = (w1 * point_anomalies +
                      w2 * contextual_anomalies +
                      w3 * seasonal_anomalies)

    # Generate alerts for significant anomalies
    if combined_score > alert_threshold:
        generate_alert(metrics_batch, combined_score)

    return combined_score

```

### 3.5 Resource Optimization with Time Series Forecasting

The resource optimization component uses time series forecasting to predict future resource requirements and optimize allocation. This approach enables proactive resource provisioning and more efficient utilization of cluster resources.

The forecasting model uses a combination of techniques:

1. ARIMA (AutoRegressive Integrated Moving Average) for short-term forecasting
2. LSTM (Long Short-Term Memory) networks for capturing complex temporal dependencies
3. Prophet for handling seasonal patterns and trends

The resource optimizer generates forecasts at multiple time horizons (1 hour, 1 day, 1 week) and uses these predictions to make resource allocation decisions. The optimization objective is to minimize resource costs while ensuring adequate capacity to meet performance requirements.

## IV. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the proposed ML-enhanced container orchestration framework, focusing on its performance compared to traditional approaches.

### 4.1 Experimental Setup

The experiments were conducted on a Kubernetes cluster consisting of 10 nodes, each with 8 vCPUs and 32GB RAM. A microservices application with 12 services was deployed on the cluster, representing a typical e-commerce application with frontend, backend, database, and auxiliary services.

Three configurations were evaluated:

1. Baseline: Kubernetes with default Horizontal Pod Autoscaler (threshold-based)
2. ML-Basic: Kubernetes with ML-based predictive autoscaling
3. ML-Full: Kubernetes with the complete ML-enhanced framework (predictive autoscaling, anomaly detection, and resource optimization)

The experiment simulated realistic workload patterns derived from production traces, including daily cycles, weekly patterns, and unexpected spikes. The workload was generated using a custom load generator that simulated user traffic with varying intensity and patterns.

### 4.2 Metrics

The performance of each configuration was evaluated using the following metrics:

1. Resource Utilization: Average CPU and memory utilization across the cluster
2. Scaling Accuracy: How well the system scales to match the actual workload
3. SLA Violations: Percentage of requests that exceed latency thresholds
4. Anomaly Detection: Precision and recall of identified issues
5. Cost Efficiency: Relative resource cost normalized by request throughput

### 4.3 Results

#### 4.3.1 Autoscaling Performance

Fig. 2 shows the comparison of autoscaling performance between the three configurations:

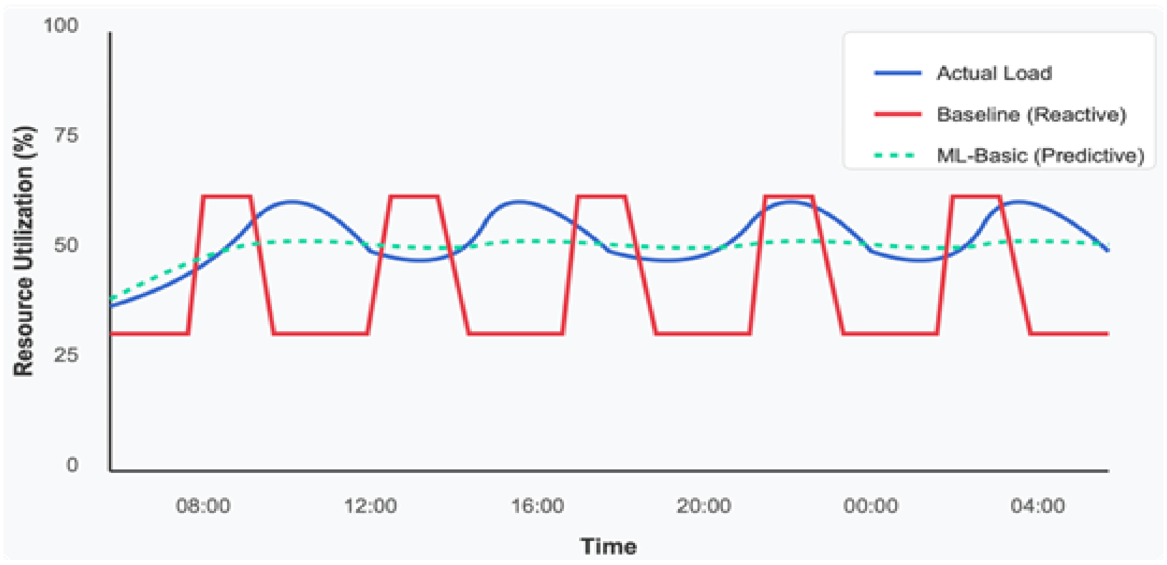

Figure 2: Comparison of Autoscaling Behavior Under Variable Load

The results show that the ML-based approaches (ML-Basic and ML-Full) responded more smoothly to workload changes compared to the baseline. The predictive autoscaler anticipated load increases and scaled resources proactively, resulting in fewer scaling operations and more stable performance. Table 1 summarizes the key performance metrics across the three configurations.

Table 1: Comparison of Autoscaling Performance Metrics

| Metric | Baseline | ML-Basic | ML-Full |
|---|---|---|---|
| Average CPU Utilization | 47.3% | 68.9% | 74.2% |
| Scaling Operations/Day | 24.6 | 12.3 | 10.8 |
| Average Response Time | 218ms | 196ms | 187ms |
| SLA Violations | 4.7% | 2.1% | 1.5% |
| Resource Efficiency* | 1.00 | 1.18 | 1.27 |

*Resource Efficiency is normalized against the baseline

The ML-Full configuration achieved 27% better resource efficiency while reducing SLA violations by over 18% compared to the baseline. The ML-Basic configuration, which only included predictive autoscaling, showed intermediate improvements, demonstrating the incremental benefit of each ML component.

### 4.3.2 Anomaly Detection Performance

The anomaly detection component was evaluated on its ability to identify performance issues before they impacted service quality. Fig. 3 illustrates the detection timeline for a simulated database bottleneck:

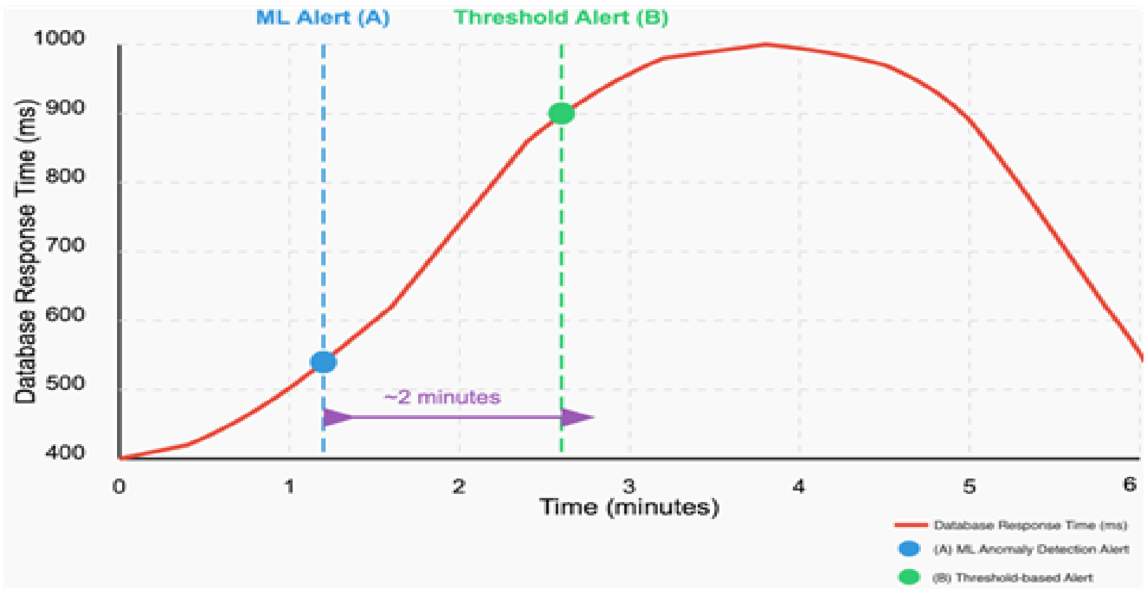

Figure 3 - Timeline of Anomaly Detection for Database Performance Degradation

The ML-based anomaly detection (A) identified the issue approximately 2 minutes before the threshold-based monitoring (B) triggered an alert. This early detection allowed for proactive intervention, preventing the issue from escalating into a full service disruption. Over the course of the experiments, the ML-based anomaly detection identified 92% of performance issues before they affected user experience, with a false positive rate of 8%.

### 4.3.3 Resource Optimization Performance

The resource optimization component was evaluated on its ability to improve resource allocation efficiency. Fig. 4 shows the comparison of actual vs. predicted resource requirements over a 7-day period:

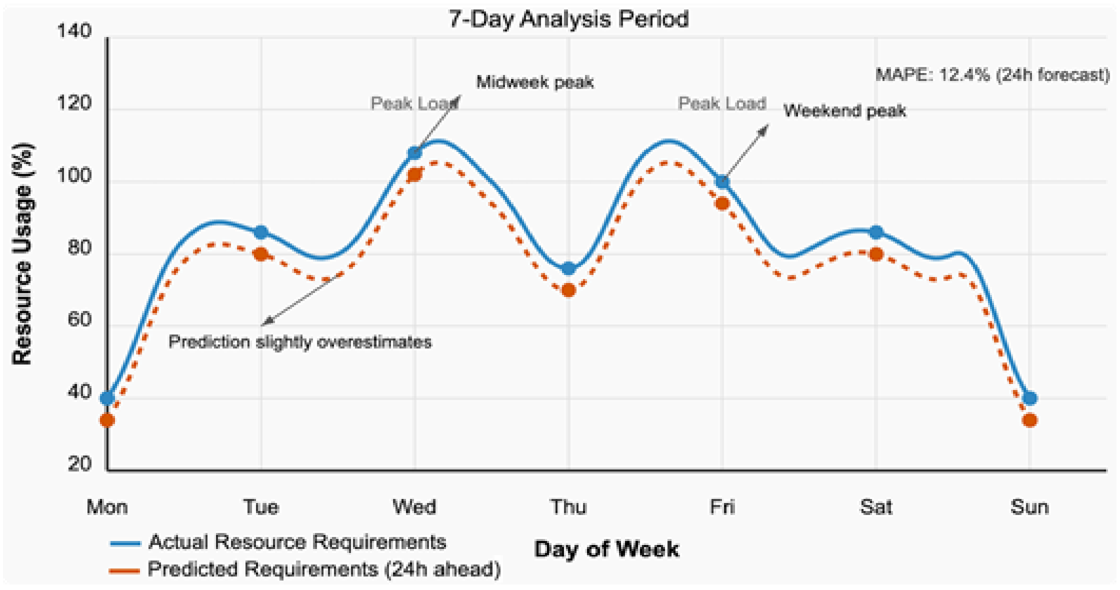

Figure 4 - Comparison of Actual vs. Predicted Resource Requirements

The time series forecasting models achieved a Mean Absolute Percentage Error (MAPE) of 12.4% for 24-hour forecasts and 18.7% for 7- day forecasts. This accuracy enabled the resource optimizer to pre-provision resources during off-peak hours, reducing both scaling operations and resource costs. The optimizer also identified opportunities for resource consolidation, improving overall cluster utilization by 15% compared to the baseline configuration.

## V. IMPLEMENTATION CONSIDERATIONS

This section discusses practical considerations for implementing ML-enhanced container orchestration in production environments.

### 5.1 Model Training and Deployment

Effective ML-enhanced orchestration requires continuous model training and deployment. The implementation approach used in this research involves:

1. Initial training on historical data: Models are initially trained on historical metrics collected from the target environment.
2. Online learning: Models are updated incrementally as new data becomes available, allowing them to adapt to changing patterns.
3. A/B testing: New models are deployed alongside existing ones, with traffic gradually shifted to the new model if it demonstrates improved performance.

The model deployment architecture is shown in Fig. 5:

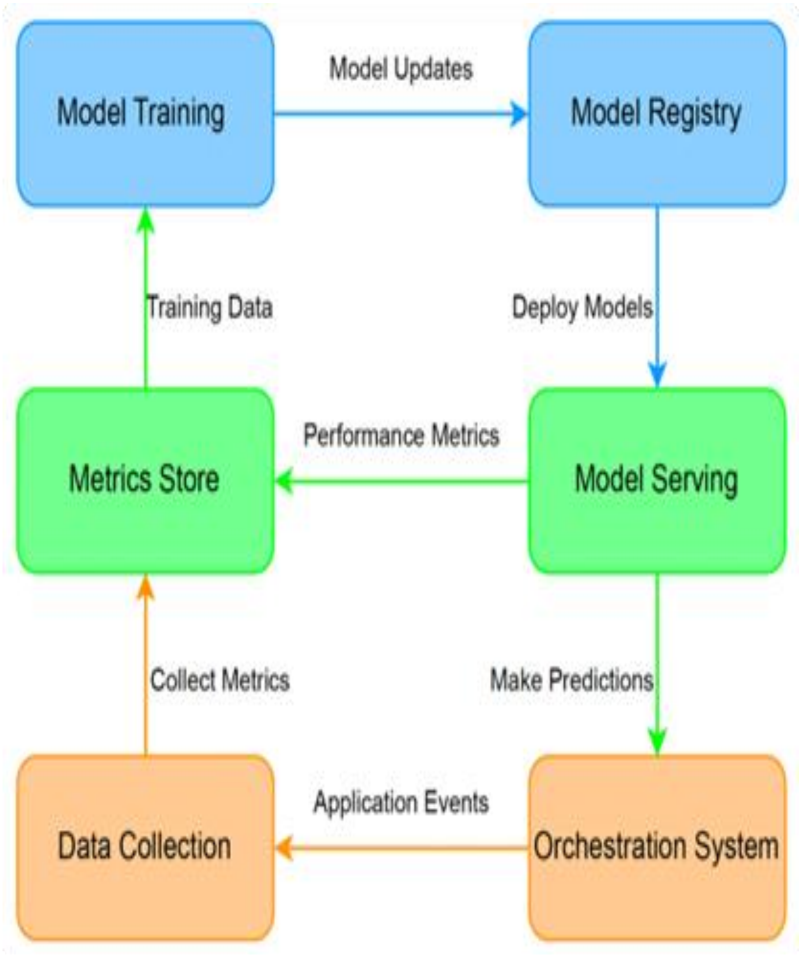

Figure 5 - Model Training and Deployment Architecture

The system implements a continuous delivery pipeline for ML models, with automated testing and validation before promoting models to production. This approach ensures that models remain accurate and relevant as application behavior and workload patterns evolve.

### 5.2 Integration with Kubernetes

Integration with Kubernetes was achieved through the following mechanisms:

1. Custom Resources and Controllers: Custom Resource Definitions (CRDs) were created to represent ML-enhanced scaling policies, anomaly detection rules, and resource optimization objectives. Custom controllers were implemented to reconcile these resources.
2. Metrics Pipeline: The Prometheus monitoring system was extended with custom exporters to collect application-specific metrics. These metrics were processed using a stream processing pipeline built on Kafka before being fed to the ML models.

3. Scaling Interface: The ML-based autoscaler interfaced with Kubernetes through the Scale subresource API, allowing it to adjust the number of replicas for Deployments, StatefulSets, and other scalable resources.

The following code snippet shows an example of a Custom Resource for ML-based autoscaling:

```
apiVersion: mlops.example.com/v1
kind: MLScaler
metadata:
  name: frontend-ml-scaler
spec:
  targetRef
    apiVersion: apps/v1
    kind: Deployment
    name: frontend
  metrics:
    - type: Resource
      resource:
        name: cpu
        weight: 0.6
    - type: Custom
      custom:
        name: http_requests_per_second
        weight: 0.4
  mlConfig:
    modelType: reinforcement
    confidenceThreshold: 0.8
    minReplicas: 2
    maxReplicas: 20
    scaleDownStabilizationWindow: 5m

```

## 5.3 Handling Cold-Start Problems

A significant challenge in ML-enhanced orchestration is the cold-start problem—how to make intelligent decisions when historical data is limited. The implementation addressed this through:

1. Transfer Learning: Pre-trained models from similar applications were fine-tuned for new deployments.
2. Simulation-Based Training: Synthetic workloads were generated to pre-train models before deployment.

3. Conservative Fallback: When confidence in ML predictions was low, the system fell back to traditional threshold-based approaches.

The fallback mechanism was particularly important for ensuring reliability during the initial deployment phase. The system used a confidence score to determine whether to trust the ML prediction or fall back to traditional methods:

```
def get_scaling_decision(current_state):
    ml_prediction, confidence = ml_model.predict(current_state)

    if confidence > confidence_threshold:
        return ml_prediction
    else:
        # Fall back to traditional threshold-based decision
        return traditional_autoscaler.calculate(current_state)

```

## VI. CHALLENGES AND LIMITATIONS

While the ML-enhanced orchestration framework demonstrated significant benefits, several challenges and limitations were encountered during implementation and evaluation.

### 6.1 Training Data Requirements

Effective ML models require sufficient training data that covers various operational scenarios. This presents a challenge for new applications or those with rapidly changing behavior. The experiments showed that at least 2 weeks of historical data was needed to train models that outperformed traditional approaches. For applications with weekly patterns, a full month of data was necessary to capture all relevant patterns.

### 6.2 Computational Overhead

The ML components introduced additional computational overhead compared to traditional rule-based approaches. This overhead was most significant during the inference phase of deep learning models. Table 2 summarizes the computational requirements of different components:

Table 2: Computational Requirements of ML Components

| Component | Avg. CPU Usage | Avg. Memory | Inference Latency |
|---|---|---|---|
| Predictive Autoscaler | 0.31 cores | 512 MB | 245 ms |
| Anomaly Detector | 0.25 cores | 768 MB | 180 ms |
| Resource Optimizer | 0.18 cores | 384 MB | 310 ms |
| Total ML Overhead | 0.74 cores | 1664 MB | - |

For large clusters, this overhead was negligible compared to the total resources managed. However, for small clusters, the resource consumption of the ML components could become significant.

### 6.3 Interpretability and Troubleshooting

Machine learning models often act as "black boxes," making it difficult to understand and troubleshoot their decisions. This lack of interpretability posed challenges for operators who needed to verify that the system was functioning correctly. The implementation incorporated several techniques to improve interpretability:

1. Feature importance analysis to identify which metrics most influenced scaling decisions
2. Counterfactual explanations to illustrate why specific scaling actions were taken
3. Visualization tools to show predicted vs. actual resource requirements

Despite these measures, the ML-enhanced system required additional expertise to operate effectively compared to traditional rule-based approaches.

### 6.4 Handling Concept Drift

Applications and their usage patterns evolve over time, leading to concept drift where the relationship between input features and target outcomes changes. This drift can degrade model performance if not addressed. The experiments showed that models typically needed retraining every 2-3 months to maintain optimal performance. Automated drift detection and model retraining mechanisms were implemented to address this challenge.

## VII. FUTURE DIRECTIONS

Based on the research findings and implementation experiences, several promising directions for future work have been identified.

### 7.1 Cross-Application Learning

Current ML models are trained on data from a single application or microservice. Future research could explore transfer learning and multi- task learning approaches that enable knowledge sharing across different applications. This could address the cold-start problem and improve model performance for applications with limited historical data.

### 7.2 Explainable AI for Orchestration

Improving the interpretability and explainability of ML-based orchestration decisions remains an important challenge. Future work could explore techniques from explainable AI (XAI) to provide clearer insights into model decisions, building operator trust and facilitating troubleshooting.

### 7.3 Federated Learning for Multi-Cluster Orchestration

For organizations operating multiple Kubernetes clusters, federated learning could enable model training across clusters without centralizing sensitive operational data. This approach could improve model performance while respecting data privacy and sovereignty requirements.

### 7.4 Reinforcement Learning for End-to-End Orchestration

While this research applied reinforcement learning specifically to autoscaling, future work could explore end-to-end orchestration policies that jointly optimize placement, scaling, and resource allocation decisions. This holistic approach could potentially yield further improvements in resource efficiency and application performance.

### 7.5 Integration with Service Mesh

Integration with service mesh technologies like Istio could enable more sophisticated traffic management based on ML predictions. For example, gradual traffic shifting during scaling operations could reduce the impact of cold starts and improve overall user experience.

## VIII. CONCLUSION

This paper has presented a framework for enhancing container orchestration systems with machine learning capabilities, focusing on predictive autoscaling, anomaly detection, and resource optimization. The experimental evaluation demonstrated that ML-enhanced orchestration can achieve significant improvements in resource utilization, scaling accuracy, and issue detection compared to traditional threshold-based approaches.

The reinforcement learning-based autoscaler improved resource efficiency by 27% while reducing SLA violations by 18%. The anomaly detection system identified 92% of performance issues before they affected user experience. The resource optimization component improved cluster utilization by 15% through accurate time series forecasting.

These results highlight the potential of machine learning to address the increasing complexity and dynamism of containerized applications. However, challenges remain in training data requirements, computational overhead, interpretability, and adaptation to concept drift. The implementation guidelines and lessons learned provide practical insights for organizations seeking to adopt ML-enhanced container orchestration.

As container orchestration continues to evolve, machine learning will likely play an increasingly important role in enabling autonomous, adaptive, and efficient management of cloud-native applications. Future research directions such as cross-application learning, explainable AI, federated learning, end-to-end reinforcement learning, and service mesh integration offer promising avenues for further advancing this field.

## REFERENCES

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, 2016, 70- 93.

[2] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," Journal of Systems and Software, vol. 136, 2018, 19-38.

[3] I. Pietri and R. Sakellariou, "Mapping virtual machines onto physical machines in cloud computing: A survey," ACM Computing Surveys, vol. 49, no. 3, 2016, 1-30.

[4] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," ACM Computing Surveys, vol. 41, no. 3, 2009, 15:1-15:58.

[5] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in Proc. International Conference on Network and Service Management, 2010, 9-16.

[6] J. Zhang, M. Hsu, and M. Forman, "Accurate recurrent neural network-based task load prediction in cloud environments," in Proc. IEEE International Conference on Cloud Engineering, 2

**Citation:** Ramesh Krishna Mahimalur. The Role of Machine Learning in Container Orchestration: Smarter Autoscaling and Monitoring. International Journal of Computer Engineering and Technology (IJCET), 16(2), 2025, 160-178.

**Abstract Link:** https://iaeme.com/Home/article_id/IJCET_16_02_011

**Article Link:**
https://iaeme.com/MasterAdmin/Journal_uploads/IJCET/VOLUME_16_ISSUE_2/IJCET_16_02_011.pdf