Addressing QoS in Kubernetes Pods Autoscaling

Jiregna Abdissa Olana[1,*,†], Maurizio Giacobbe[2,*,†], Sarah Zanafi[1,*,†] and Antonio Puliafito[1,3]

[1]Department of Engineering, University of Messina, Contrada di Dio, Messina, 98158, Italy
[2]Dept. of Math., Computer, Physical and Earth Sciences, University of Messina, Viale Ferdinando Stagno d'Alcontres 31, Messina, 98166, Italy
[3]Consorzio Interuniversitario Nazionale per l'Informatica (CINI), via Ariosto 25, Rome, 00185, Italy

Abstract

The Kubernetes open source system is a "de facto" standard for automating deployment, scaling, and management of containerized applications. However, its static approach to QoS classifications (Guaranteed, Burstable, BestEffort) and default autoscalers (HPA, KPA) often fall short in large scale and high dynamic applications. Key limitations include lack of application awareness, reliance on limited metrics, latency in scaling actions, and poor handling of cold starts. Furthermore, incorrect configuration of resource requests and overcommitment can lead to inefficient scaling and service degradation. These challenges highlight the need for adaptive, application-aware autoscaling strategies and precise resource provisioning to ensure reliable QoS under resource contention. To address these challenges, predictive autoscaling emerges as a promising direction—leveraging historical data and machine learning techniques to anticipate workload patterns and proactively adjust resources before performance issues arise. This work examines current limitations in Kubernetes autoscaling and outlines future directions, emphasizing adaptive, application-aware, and predictive strategies for more efficient and reliable resource provisioning under dynamic workloads and resource contention.

Keywords

Quality of Service (QoS), Kubernetes, Predictive Autoscaling, Service Level Objectives (SLOs), Artificial Intelligence (AI),

1. Introduction

Kubernetes [1], commonly called *K8s*, is an open-source container orchestration platform that has become the *de facto* standard for deploying, managing, and scaling containerized applications in edge-cloud environments. It is suitable for on-premises, hybrid, or public cloud infrastructure and for a wide range of workloads, from microservices and web applications to machine learning pipelines and real-time data processing. In a Kubernetes environment, a **Pod** is the smallest (i.e., the fundmental) unit for deploying and managing containerized applications. Each Pod contains a single application instance and can hold one or more containers. Kubernetes manages Pods as part of a deployment and can perform vertical or horizontal scaling as needed. In such a context, **Quality of Service (QoS)** is a mechanism for prioritizing resource allocation among Pods based on their resource requirements and usage. In real-world scenarios, the Pod workload varies based on the characteristics of each service [2]. To ensure performance isolation and efficient resource sharing, Kubernetes categorizes Pods into QoS classes: *Guaranteed*, *Burstable*, and *BestEffort*, based on their resource requests and limits.

Autoscaling in Kubernetes, enabled by default components such as the Horizontal Pod Autoscaler (HPA) and Kubernetes-based Event-Driven Autoscaler (KEDA), allows workloads to adjust to changing demands. However, these mechanisms typically rely on basic metrics like CPU and memory usage and operate without a deep understanding of application-specific performance goals. This can result in suboptimal scaling behavior, especially in dynamic and latency-sensitive scenarios such as e-commerce

QualITA 2025: The Fourth Conference on System and Service Quality, June 25 and 27, 2025, Catania, Italy
*Corresponding author.
†These authors contributed equally.
jiregnaabdissa.olana@unime.it (J. A. Olana); mgiacobbe@unime.it (M. Giacobbe); szanafi@unime.it (S. Zanafi); apuliafito@unime.it (A. Puliafito)
0009-0002-2220-1359 (J. A. Olana); 0000-0001-6178-7132 (M. Giacobbe); 0000-0002-0126-7837 (S. Zanafi); 0000-0003-0385-2711 (A. Puliafito)

platforms during peak traffic, video streaming services under load, or edge computing applications with constrained resources [3].

The weaknesses of reactive autoscaling techniques have been more evident in recent years, particularly in systems where availability, throughput, or latency are crucial. Scaling-up is usually only initiated by reactive ways after performance decline is identified, which frequently leads to delayed reaction. Predictive autoscaling techniques [4], which change the paradigm from reaction to anticipation, have gained popularity as a result. Predictive autoscaling allows systems to anticipate future demand and make necessary resource adjustments using historical trends, real-time, and sophisticated forecasting models. This change enhances the general effectiveness and stability of native cloud deployments in addition to helping to meet strict service-level objectives (SLOs). Predictive techniques provide a possible path for next-generation orchestration and resource management in edge-cloud systems, where workloads might be bursty and delay-sensitive. It represent a really promising direction for the future.

This paper addresses the challenges in current Kubernetes autoscaling with respect to maintaining QoS guarantees and highlights the need for adaptive, application-aware strategies that can better align with service-level objectives and workload characteristics.

2. Related Works

The increasing proliferation of time-sensitive, large-scale applications and services needs scalable and low-latency infrastructures capable of meeting stringent performance requirements. In hybrid edge-cloud architectures [5], autoscaling has become a vital methodology for dynamic resource management under varying workload conditions. The impact concerns also IoT networks because they need to support dynamic scalability, thus to adapt the workload in real time to varying data volumes and computational demands [6]. Traditional autoscaling approaches are predominantly reactive, initiating scaling actions based on static thresholds of system metrics such as CPU utilization, memory usage, or response latency. These methods often suffer from delayed reactions to workload changes, thus resulting in two major issues: (i) under-provisioning, where insufficient resources lead to service degradation, and (ii) over-provisioning, where excessive resource allocation causes unnecessary operational costs. To mitigate these drawbacks, **predictive autoscaling** utilizes workload forecasting to enable proactive and efficient scaling decisions.

In [7] a predictive Decision Tree Regression (DTR) policy is used for Kubernetes vertical pod autoscaling (VPA). Although the approach is a typical example of reactive method applied to a Kubernetes architecture, it is useful to understand the importance of balancing VPA and HPA in a coordinated manner to avoid over-provisioning and resource bottlenecks.

A cost-optimized predictive autoscaling model that integrates queuing theory and game theory to enhance cloud resource management is presented in [8]. Using M / M / c and M / G / 1 queuing models, the approach dynamically adjusts resource allocation based on workload variations and significantly reduces over-provisioning costs by up to 30%, while maintaining a low request waiting time below 100 ms in peak scenarios.

A predictive autoscaling framework, leveraging ML techniques to anticipate and proactively adjust the resources allocated to containerized applications, is discussed in [4]. Results highlight the framework is a good starting point to maintain QoS within defined thresholds.

A novel graph-based proactive HPA strategy for microservices using long short-term memory (LSTM) and graph neural network (GNN) based prediction methods, namely Graph-PHPA, is presented in [9]. The proposed model is specifically designed to predict vCPU utilization, incorporating workload characteristics as its main input attribute.

Moreover, it is also important to guarantee specific SLOs for the IoT applications running on the "fog" micro datacenters (i.e., at the architectural layer between the edge and the cloud). The automatic scaling of allocated resources by efficiently utilizing the available infrastructure capacity is mandatory for QoS. A novel predictive autoscaling method for the microservice-based application hosted in a containerized fog computing infrastructure is presented in [10]. The method uses a forecasted workload to identify

the number of containers required to serve the workload with minimal response time SLOs violations. Specifically, the authors used two benchmark applications to emulate the CPU-bound workload: the fast Fourier transform (FFT) application, and the ML application. A regression-based supervised machine learning algorithm (SVR) has been implemented to predict the next time interval of temperature using historical access logs collected from the sensors.

3. QoS-Aware Autoscaling: Background and Challenges

Autoscaling in Kubernetes faces several significant challenges that can impact the system's overall performance and efficiency. One of the primary concerns is **latency in scaling decisions**. Autoscalers may not always react quickly enough to sudden spikes in workload, which can lead to a degradation in performance. This delay in scaling actions makes it harder to maintain consistent service quality under fluctuating demands.

Another issue that arises is **resource over-provisioning or under-provisioning**. Many scaling decisions are based on narrow metrics such as CPU utilization or concurrency, which do not always provide a complete picture of the system's needs. This can result in either excessive resource allocation, leading to wasted capacity [11], or insufficient resource allocation, which can cause performance bottlenecks and inefficiencies.

Additionally, **cold start latency** is a significant factor. When scaling down to zero replicas, the need to reinitialize Pods introduces startup delays. These delays can affect response times and, consequently, user experience, especially during periods of sudden traffic.

The configuration of resource *requests* and *limits* plays a critical role in shaping autoscaling behavior. When these configurations are suboptimal, several issues can arise. For instance, **evictions** occur when Pods exceed their allocated resource limits, leading to terminations that affect service availability. Similarly, **inefficient resource utilization** can happen when requests are overestimated, leading to wasted resources, or when requests are underestimated, causing resource contention. These misconfigurations also lead to **scaling inefficiencies**, as autoscalers may struggle to make timely and appropriate scaling decisions.

Another significant challenge lies in the lack of **application-specific insights** in current autoscaling strategies. Without understanding the application's unique performance characteristics, scaling decisions are often made based on generic metrics. This misalignment can result in **suboptimal scaling decisions**, where autoscalers fail to account for nuances in the application's behavior, thus failing to meet its specific requirements. This can also lead to an **inability to meet SLOs**, as the autoscaling mechanisms may not be aware of critical application requirements.

Finally, the practice of **overcommitting** resources—setting requests significantly lower than actual usage—can have serious consequences. This often leads to **resource contention**, where multiple Pods compete for the same insufficient resources, degrading overall system performance. Moreover, **frequent evictions** may occur as the system struggles to handle resource pressure, and **unpredictable scaling** becomes a norm, as overcommitment complicates the autoscaler's ability to make accurate predictions and plan resource allocation effectively.

4. Research Directions and Proposed Approach

Table 1 highlights the core differences between Kubernetes' current static resource management approach and a proposed dynamic, AI-enabled alternative. Static approach lacks flexibility and does not account for dynamic workload behavior or application-specific requirements [12]. In contrast, the proposed approach introduces intelligent autoscaling mechanisms that continuously adapt resource allocations based on live metrics, predicted load, and SLOs. Techniques such as reinforcement learning (RL)[13, 14, 15] and time-series forecasting [16, 17] can be employed to anticipate traffic patterns and proactively scale applications before resource bottlenecks occur. In particular, RL adjusts resources by

learning from ongoing interactions with the application environment. Over time, decisions are refined based on how well they meet SLOs while optimizing resource utilization.

Hybrid models [18] emerged to combine forecasting with RL by feeding predictive data into the learning agents. This combination improves both the precision and the consistency of autoscaling decisions, leveraging forecasts for short-term accuracy and the adaptability of RL for long-term optimization. This combination balances forecast accuracy with long-term optimization power of RL.

Moreover, integrating application-level or custom [19] metrics (e.g., latency, request rate, error rate) enables more fine-grained and QoS-aligned scaling decisions. The shift from reactive, threshold-based rules to adaptive and predictive policies represents a fundamental challenge in modern cloud-native systems. By adopting AI-based models, Kubernetes can evolve into a more resilient and efficient platform capable of meeting the demands of dynamic and complex workloads.

4.1. Ensuring QoS Through Predictive and Proactive Autoscaling

Maintaining Quality of Service (QoS) in cloud-native and latency-sensitive applications requires to prevent SLA violations, especially under dynamic and unpredictable workloads. To address this limitation, in our approach we integrate both *predictive* and *proactive* strategies. Although these terms are related, they represent distinct components in an effective QoS-oriented architecture.

Predictive autoscaling involves the use of forecasting models (e.g., time series analysis, statistical learning, or ML) to anticipate future resource demands based on historical and real-time telemetry data. These models enable systems to estimate upcoming workload intensities or performance bottlenecks with varying degrees of accuracy.

Proactive autoscaling, instead, refers to the system's ability to act in advance of anticipated changes. It translates predictive insights into timely resource provisioning actions. For example, if a model predicts a significant increase in request traffic within the next interval, a proactive policy may scale out the application pods before the traffic spike occurs, thereby preserving response time and system stability.

The combination of predictive foresight with proactive execution is critical for QoS preservation. Predictive models alone offer valuable insights, but without timely action, they do not mitigate performance degradation. On the other side, without accurate forecasting, proactive actions risk unnecessary resource allocation or delayed responses. Together, these mechanisms enable a more intelligent and adaptive scaling behavior that minimizes latency, maintains throughput, and reduces the likelihood of SLA violations.

Therefore, the joint application of predictive and proactive techniques constitutes a robust strategy for QoS-aware autoscaling in modern cloud environments.

4.2. Proposed AI-driven Predictive and Proactive Autoscaling System

A conceptual model on the integration of a **predictive autoscaler** based on AI/ML technologies is shown in Figure 1. In particular, the `AI/ML Predictive Autoscaler` is designed as part of a hybrid scenario in which proactive policies are deployed.

It represents the starting point for the evolution of the Kubernetes static QoS model to a dynamic AI-driven proactive autoscaling system.

We expand on the previously presented system by categorizing application types according to their requirements for autoscaling and connecting them to suitable prediction methods according to operational objectives and application logic. Because it ignores workload unpredictability and temporal shifts, current Kubernetes autoscaling, which is usually reactive and linked to static CPU/memory thresholds, frequently fails in dynamic contexts. We suggest a versatile autoscaling technique that makes use of prediction models derived from both historical and real-time data in order to get around this. This makes proactive resource provisioning possible, predicting demand before performance declines. In addition, the architecture supports particular performance objectives such as low latency, fast throughput, and lower error rates. Two major problems with traditional autoscaling are that it

Table 1
From Static QoS to Dynamic, AI-Driven Autoscaling in Kubernetes

Aspect	Static Approach (Current Kubernetes)	Dynamic/Intelligent Approach (Proposed)
Main Challenge	Static QoS and scaling policies do not reflect dynamic service requirements	Evolve toward dynamic, intelligent autoscaling systems that align resource allocation with real-time demands
QoS Classification	Three fixed QoS classes (*Guaranteed*, *Burstable*, *BestEffort*) based on static resource requests/limits	Fine-grained, context-aware QoS levels inferred from real-time workload and system metrics
Autoscaling Logic	Threshold-based policies (e.g., CPU > 80%) drive decisions with limited adaptability	Predictive, AI-driven policies using ML models (e.g., time series, RL) to anticipate workload changes
Resource Awareness	Blind to runtime behavior; resources allocated statically at deployment time	Continuously adjusted based on observed usage patterns and workload characteristics
Application Awareness	Lacks integration with application-specific metrics or SLOs	Incorporates custom application metrics (e.g., latency, errors, throughput) to guide scaling
Cold Start and Scaling Latency	No proactive scaling; user experience impacted during traffic surges	AI-enabled pre-scaling and warm-up strategies based on predicted load
Adaptivity	Manual tuning; reactive to metric breaches	Self-adaptive through feedback loops; auto-tuning of scaling parameters over time

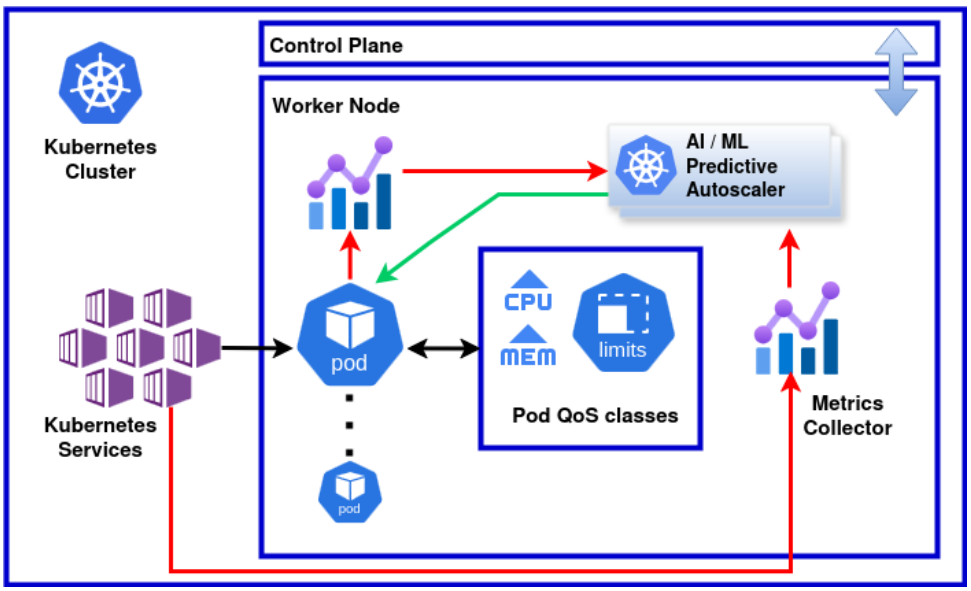

Figure 1: Evolution of the Kubernetes static QoS model to a predictive and proactive, AI-driven autoscaling system.

cannot predict workload trends and only responds when predetermined thresholds are crossed. These shortcomings are particularly critical in latency-sensitive systems, where even minimal delays can adversely affect performance and violate Service Level Agreements (SLAs), ultimately degrading user experience. In contrast, predictive autoscaling approaches, which employ forecasting models, enable proactive resource allocation. By anticipating workload increases, these techniques facilitate smoother

system performance and ensure that additional resources are provisioned in advance of actual demand.

Different scaling needs characterize applications: Table 2 shows main categories according to their scaling sensitivity and appropriate forecasting techniques. By categorizing applications based on their behavior, it becomes possible to adapt autoscaling strategies accordingly to the performance needed.

Table 2
Autoscaling Needs by Application Type

Application Type	Scaling Needs	Recommended Techniques	Reference
Latency-sensitive (e.g., video streaming, gaming)	Rapid scaling with minimal startup delay to prevent lag	Bi-LSTM forecasting with trend/anomaly detection	Dang-Quang & Yoo, 2022 [17]
Batch / ML workloads	Optimize throughput under deadlines while reducing cost	RL-based or deadline-aware scheduling	Garí et al., 2020 [15]
IoT / Edge computing	Lightweight models that handle sudden bursts on resource-limited devices	Hybrid edge/cloud forecasting with heuristics	Pahlavanpour et al., 2021 [18]
Microservices / API	Quick, metric-driven adjustments to meet fine-grained SLOs	Adaptive multi-metric controllers with thresholds	Nunes et al., 2024 [19]
Maintaining Quality of Service (QoS) in cloud-native and latency-sensitive applications	Reduce latency, avoid SLOs violations, minimize over-provisioning, and improve user satisfaction	Hybrid predictive/proactive approach	**This Work**

5. Case Study: Machine Learning Inference Services Necessitating Low-Latency Autoscaling

Contemporary applications utilizing Artificial Intelligence and the Internet of Things within the AIoT paradigm increasingly depend on real-time inference from implemented machine learning models, hence providing dynamic user experiences. Services must meet strict SLOs, especially for latency and availability. Moreover, reactive threshold-based scaling or static provisioning frequently results in over-provisioning during idle times or under-provisioning during demand surges. Traditional Kubernetes autoscalers introduce latency between traffic increase and scaling response, resulting in: (i) cold-start penalties for containerized inference runtimes (e.g., TensorFlow Serving, TorchServe), (ii) latency violations when inference requests queue up, and (iii) inefficient use of GPU/CPU resources when demand prediction is misaligned with scale actions.

By using a learned forecasting model to estimate load and proactively modify the number of inference service replicas, our AI-based predictive autoscaling method overcomes these drawbacks. Our predictive approach reduces tail latency (P95, P99) violations during peak periods and allows the proactive distribution of replicas to handle anticipated load spikes with little cold-start delay. The time limits below which 95% and 99% of all requests are fulfilled are denoted by the P95 and P99 latencies, respectively. P99 represents high tail delay, whereas P95 represents the usual worst-case latency encountered by customers. These metrics are essential for comprehending and improving a system's responsiveness and dependability under load. Monitoring tail latencies helps ensure end-to-end QoS, especially for time-sensitive tasks like health monitoring or industrial automation.

Table 3
Software Testbed Configuration

Component	Details / Version
Container Orchestration	Minikube 1.35.0
Container Runtime	Docker 28.3.1
Kubernetes CLI	Kubectl 1.32.0
Scripting Environment	Python 3.12.7
Package Manager	Helm 3.18.2
Metrics Collection	`kubectl logs` for Replica Count and Tail Latencies (P95, P99)

5.1. Design and Configuration of the Software Testbed

To evaluate the behavior and performance of the autoscaling policies under realistic workload conditions, a controlled and reproducible testbed environment was configured using widely adopted, production-relevant tools and platforms. The selected stack ensures compatibility with modern cloud-native practices while maintaining lightweight and deterministic deployment characteristics for local experimentation.

5.1.1. Minikube 1.35.0 for Container Orchestration

Minikube was selected due to its ease of use and capacity to replicate a fully functional Kubernetes environment on a single local computer. Version 1.35.0 provides compatibility with recent Kubernetes features, including autoscaling APIs and metrics server support, while avoiding the complexity of managing a multi-node cluster. When creating unique autoscaling logic, this configuration facilitates quick iteration and debugging.

5.1.2. Container Runtime: Docker 28.3.1

Minikube's underlying container runtime, Docker, allows for standardized application workload execution and packaging. It is indicated for modeling real-world deployments and container behavior, including resource isolation and cold start latency impacts, due to its maturity, reliability, and wide ecosystem support.

5.1.3. Kubernetes CLI - Kubectl 1.32.0

Kubectl was used to interface with the cluster, install services, monitor resources, and extract important telemetry. In order to prevent compatibility problems and guarantee accurate diagnostics during testing, version 1.32.0 guarantees alignment with the Kubernetes API level supported by Minikube 1.35.0.

5.1.4. Scripting Environment: Python 3.12.7

The latency modeling logic and autoscaling simulation were implemented in Python. Python's versatility made it ideal for quickly prototyping both heuristic and predictive scaling methods, and the 3.12.7 version boasts recent performance and syntactic enhancements. Additionally, thorough analysis and visualization of system behavior were made possible by Python's scientific libraries, such as NumPy and Matplotlib.

5.1.5. Helm 3.18.2 Package Manager

Helm was used to package and administer Kubernetes deployments, guaranteeing uniformity between runs and streamlining configuration modifications. Repeatable experiments required modular definitions of services, metrics collectors, and scaling setups, which were made possible by its templating capabilities.

5.1.6. Metrics Collection – kubectl logs for Replica Count and Tail Latencies (P95, P99)

Kubectl logs were used to gather system metrics, and application-level output was parsed to retrieve tail latency statistics and replica counts, with an initial focus on P95 latency. The simplicity, low overhead, and capacity to offer fine-grained understanding of temporal system behavior without the need for an external telemetry stack made this approach the preferred choice. The P95 metric is commonly used as a useful performance indicator in mild tail situations, since it measures the latency 95% of the experience of requests. It was chosen as the starting point for comparison because it provides statistically reliable insights even in the presence of fluctuating traffic. However, the testbed was later expanded to collect P99 latency as well in order to gain a better understanding of extreme tail behavior. P99 is essential in latency-sensitive applications, especially in IoT or edge computing scenarios where even a small percentage of delayed requests can affect control loops or sensor-actuator responsiveness, even though it is more sensitive to outliers and usually requires larger datasets for statistical significance.

5.2. Experimental Setup and Evaluation Procedure

Algorithm 1 Predictive and Reactive Autoscaling Algorithm

Require: Request rate trace $QPS[0..T-1]$, threshold θ, lookahead L, window W
Ensure: Replica counts $R_{HPA}[0..T-1]$, $R_{Pred}[0..T-1]$

Initialize $R_{Pred}[0] \leftarrow 1$, $R_{HPA}[0] \leftarrow 1$
for $t = 0$ to $T-1$ **do**
  // **Reactive HPA**
  $avg \leftarrow \text{mean}(QPS[\max(0, t-W)..t])$
  for $n = 1$ to n_{max} **do**
    $lat \leftarrow \text{EstimateLatency}(avg, n)$
    if $lat \leq \theta$ **then**
      $R_{HPA}[t] \leftarrow n$
      break
    end if
  end for
  // **Predictive Scaling**
  $future \leftarrow \text{mean}(QPS[t+1..t+L])$
  for $n = 1$ to n_{max} **do**
    $lat_{curr} \leftarrow \text{EstimateLatency}(QPS[t], n)$
    $lat_{future} \leftarrow \text{EstimateLatency}(future, n)$
    if $lat_{curr} \leq \theta$ **and** $lat_{future} \leq \theta$ **then**
      $target \leftarrow n$
      break
    end if
  end for
  // Apply hysteresis
  $R_{Pred}[t] \leftarrow \min(R_{Pred}[t-1]+1, \max(R_{Pred}[t-1]-1, target))$
end for=0

As demonstrated in the Algorithm 1, we constructed and contrasted reactive HPA and a lookahead-based predictive scaler to assess autoscaling behavior under various workload scenarios. Figure 2 schema of the testbed.

By scaling the number of copies according to a brief history of request rate measurements (such as a 3-minute average), the reactive method mimics conventional HPA activity. Using a simple queuing model that incorporates cold-start latency, per-replica processing capacity, and overload penalty, the scaler repeatedly calculates the estimated latency for a specified number of replicas. Choose the fewest replicas necessary to keep the estimated latency below a predetermined threshold.

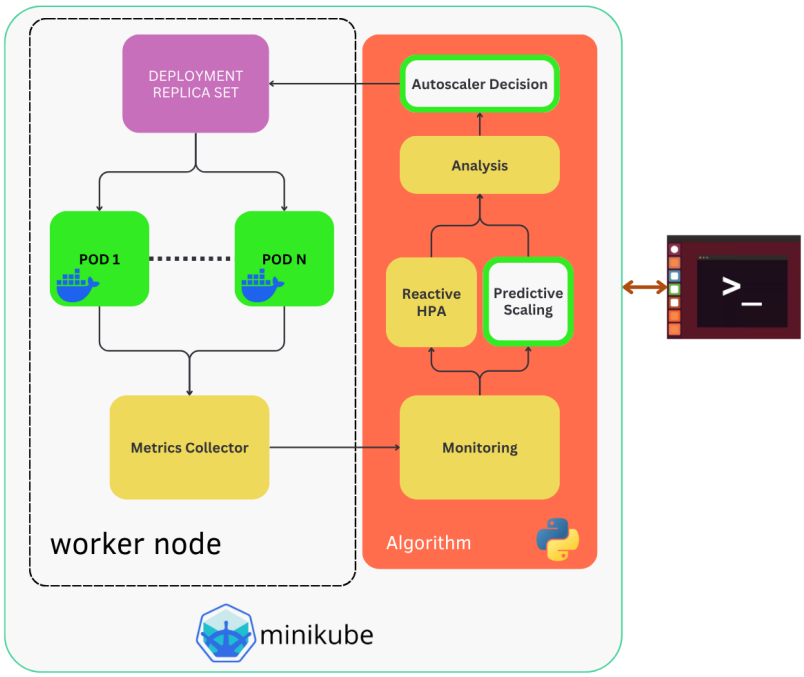

Figure 2: Experimental testbed architecture for predictive and reactive autoscaling evaluation.

By adding a short-term forecast based on a set lookahead window, the predictive strategy expands on this methodology. It determines the anticipated request rate for the near future for every timestep and makes sure that the predicted and actual latency values stay below the threshold before scaling. Through proactive capacity provisioning prior to increases, this helps avoid threshold violations. By restricting the number of clones that can be eliminated in a single step, a hysteresis mechanism is used to avoid frequent scale-downs.

A simulated workload trace with two separate traffic peaks and Gaussian noise was used to test the technique. The predictive scaler's efficacy was evaluated using metrics like P95 and P99 delay, violation counts, and relative improvement over the reactive baseline.

5.3. Experimental Results

Figure 3 illustrates the QPS input pattern, the corresponding latency measurements, and replica provisioning dynamics for both HPA and predictive strategies. These results demonstrate that anticipatory autoscaling using lightweight predictive logic can substantially improve SLOs adherence under dynamic workloads without over-provisioning resources.

Table 4
Tail-latency violation comparison between HPA and predictive autoscaling.

Metric	HPA	Predictive (Improved)
P95 Violations	7	0
P95 Latency (ms)	330.73	192.51 (41.79%)
P99 Latency (ms)	427.49	196.68 (53.99%)

The comparative results between the baseline HPA strategy and the improved predictive autoscaler are summarized in Table 4. The predictive policy reduced the number of latency violations of P95

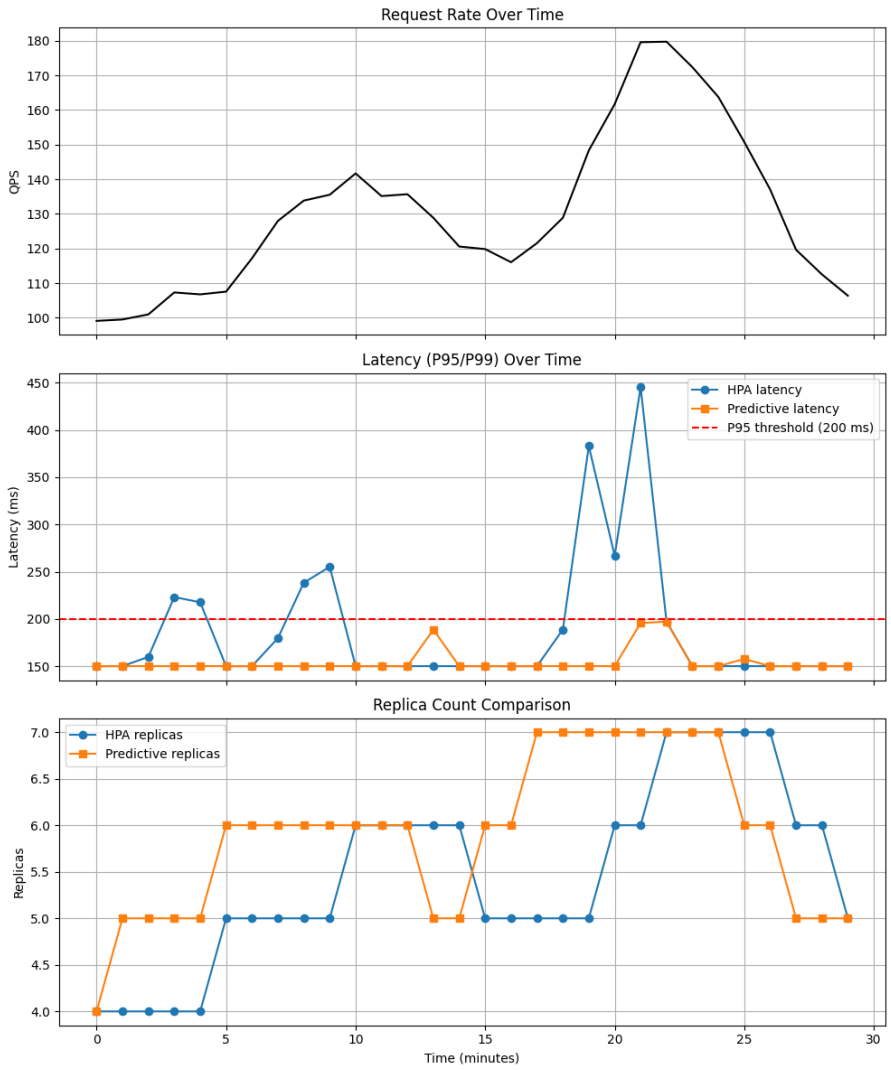

Figure 3: Simulation of QPS load, tail latency, and replica count over 30 minutes. The predictive autoscaler anticipates demand and reduces latency violations.

from 7 to 0. Furthermore, the latency of P95 and P99 was, respectively, reduced by 41.79% and 53.99%, demonstrating improved robustness under peak load. The latency is now below the threshold of 200 ms.

The improvements are particularly critical in IoT environments, where responsiveness and predictability are essential. By minimizing tail latencies, the predictive autoscaler ensures consistent QoS even during traffic bursts, which is vital for time-sensitive applications such as real-time monitoring, anomaly detection, or actuation in smart systems. Furthermore, the elimination of latency violations supports the enforcement of strict SLOs commonly required in IoT deployments. The use of P95 and P99 metrics allows for fine-grained control and visibility over system performance, helping to guarantee reliable communication and low-latency feedback loops across distributed devices and edge-cloud architectures.

These results strongly suggest that edge processing, when coupled with an intelligent and predictive autoscaling strategy, is more effective in meeting stringent latency requirements, especially in IoT scenarios. For edge and IoT workloads, where delayed responses can deteriorate system behavior, zero P95 violations under the predictive strategy translates into dependable real-time performance. The system's ability to manage worst-case spikes, which frequently occur unexpectedly in edge contexts (such as motion sensors or alarms), is demonstrated by the remarkable reduction in P99 latency (53.99%). Because the autoscaler is predictive and anticipates load rather than reacting slowly like HPA, the improvement is achieved without excessive over-provisioning.

6. Conclusions and Future Work

This paper introduced a dynamic and intelligent approach aimed at enhancing QoS in Kubernetes-based environments. We proposed a conceptual framework that emphasizes the need for greater awareness of applications and resources, adaptive scaling behavior, and proactive handling of challenges such as cold starts and scaling latency. Our model highlights the importance of transitioning from static threshold-based autoscaling to more context-aware and predictive mechanisms. These findings imply that reactive scaling (HPA) in a cloud-only architecture might not be adequate for use cases that are sensitive to latency. A hybrid or edge-first strategy with predictive autoscaling can provide noticeably reduced and more stable tail latencies, guaranteeing QoS compliance even under erratic or bursty loads that are common in IoT systems.

Looking ahead, future work will focus on the exploration and implementation of new AI-driven autoscaling strategies. In particular, the integration of machine learning (ML) and reinforcement learning (RL) techniques holds significant promise in addressing the limitations of current approaches. These intelligent methods can leverage real-time and historical performance data to predict workload trends and optimize resource provisioning decisions accordingly. By enabling proactive scaling actions, such models have the potential to maintain application QoS even under highly dynamic workload conditions. Furthermore, ongoing research should investigate the design of feedback control loops, continuous learning mechanisms, and multi-metric optimization strategies to ensure that autoscaling policies align not only with system-level metrics but also with service-level objectives (SLOs). Finally, experimentation in real-world Kubernetes deployments and benchmarking against existing reactive solutions will be essential to validate the effectiveness and reuse of the proposed approach.

Acknowledgments

This work was partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, project SecCO, CUP D33C22001300002, and project 3D-SEECSDE, CUP J33C22002810001, partnership on “SEcurity and RIghts in the CyberSpace” (PE00000014 - program “SERICS”).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Kubernetes Authors, Kubernetes Documentation, https://kubernetes.io/, 2025. Accessed: 2025-05-13.

[2] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, J. Song, Research on resource prediction model based on kubernetes container auto-scaling technology, IOP Conference Series: Materials Science and Engineering 569 (2019) 052092. URL: https://dx.doi.org/10.1088/1757-899X/569/5/052092. doi:10.1088/1757-899X/569/5/052092.

[3] J. Cámara-Miró, L. Costero, F. D. Igual, Qos-aware workload scheduling on heterogeneous and dynamic edge-to-cloud deployments, in: Proceedings of the 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), IEEE, Turin, Italy, 2025, pp. 321–328. doi:10.1109/PDP66500.2025.00052.

[4] A. K. Mogal, V. P. Sonaje, Predictive autoscaling for containerized applications using machine learning, in: 2024 1st International Conference on Cognitive, Green and Ubiquitous Computing (IC-CGU), 2024, pp. 1–6. doi:10.1109/IC-CGU58078.2024.10530773.

[5] M. Giacobbe, I. Falco, S. Zanafi, C. Colarusso, J. A. Olana, A. Puliafito, E. Zimeo, Key challenges in lorawan-based edge-cloud infrastructures for security-sensitive smart cities applications,

volume 3962, 2025. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105005832558&partnerID=40&md5=9073ecd009e69ada004132c8cf7ff041.

[6] I. Ficili, M. Giacobbe, G. Tricomi, A. Puliafito, From sensors to data intelligence: Leveraging iot, cloud, and edge computing with ai, Sensors 25 (2025). URL: https://www.mdpi.com/1424-8220/25/6/1763. doi:10.3390/s25061763.

[7] Z. Bouflous, F. Haraka, M. Ouzzif, K. Bouragba, Enhanced vertical pod auto scaling with decision tree regressor-max in kubernetes, in: 2024 11th International Conference on Wireless Networks and Mobile Communications (WINCOM), 2024, pp. 1–5. doi:10.1109/WINCOM62286.2024.10654970.

[8] V. Pandey, Cost-optimized predictive autoscaling of cloud resources using game-theoretic queuing theory, in: SoutheastCon 2025, 2025, pp. 481–487. doi:10.1109/SoutheastCon56624.2025.10971700.

[9] H. X. Nguyen, S. Zhu, M. Liu, Graph-phpa: Graph-based proactive horizontal pod autoscaling for microservices using lstm-gnn, in: 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), 2022, pp. 237–241. doi:10.1109/CloudNet55617.2022.9978781.

[10] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, A. Erradi, Predictive autoscaling of microservices hosted in fog microdata center, IEEE Systems Journal 15 (2021) 1275–1286. doi:10.1109/JSYST.2020.2997518.

[11] D.-D. Vu, M.-N. Tran, Y. Kim, Predictive hybrid autoscaling for containerized applications, IEEE Access 10 (2022) 109768–109778. doi:10.1109/ACCESS.2022.3214985.

[12] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. V. Mayoral, F. S. Tehàs, Autoscaling pods on an on-premise kubernetes infrastructure QoS-aware, IEEE Access 10 (2022) 33083–33094. doi:10.1109/ACCESS.2022.3158743.

[13] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource management with deep reinforcement learning, in: Proceedings of the 15th ACM Workshop on Hot Topics in Networks, ACM, 2016, pp. 50–56.

[14] A. A. Khaleq, I. Ra, Development of qos-aware agents with reinforcement learning for autoscaling of microservices on the cloud, in: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), 2021, pp. 13–19. doi:10.1109/ACSOS-C52956.2021.00025.

[15] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, C. G. Garino, Reinforcement learning–based application autoscaling in the cloud: A survey, arXiv preprint arXiv:2001.09957 (2020). URL: https://arxiv.org/abs/2001.09957.

[16] W. Miao, Y. Sun, Z. Zeng, T. Hong, M. Xiao, An container elastic autoscaling strategy based adaptive integrated resource forecast, in: 2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE), 2024, pp. 525–530. doi:10.1109/CISCE62493.2024.10653143.

[17] N. Dang-Quang, M. Yoo, An efficient multivariate autoscaling framework using bi–lstm for cloud computing, Applied Sciences 12 (2022) 3523. URL: https://www.mdpi.com/2076-3417/12/7/3523. doi:10.3390/app12073523.

[18] N. Pahlavanpour, A. Jan, J. Ahson, Proactive Pod Autoscaler (PPA) for Kubernetes-based Edge Computing Applications, Technical Report, DIVA Repository, 2021. URL: https://www.diva-portal.org/smash/get/diva2:1595932/FULLTEXT01.pdf.

[19] J. P. K. S. Nunes, S. Nejati, M. Sabetzadeh, E. Y. Nakagawa, Self-adaptive, requirements-driven autoscaling of microservices, arXiv preprint arXiv:2403.08798 (2024). URL: https://arxiv.org/pdf/2403.08798.