International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Predictive Auto Scaling and Cloud Cost Optimization Using Machine Learning

Author: **Kirti Patole, Pranjali Kale, Sajjan Barbade, Mahadev Shete, Pranita Manjare**
Dr. D. Y. Patil Institute of MCA and Management, Akurdi, Pune

ABSTRACT

Cloud applications experience fluctuating demand, requiring resource controllers to balance infrastructure cost against service performance. Reactive autoscaling changes capacity after monitored utilization crosses a threshold; predictive autoscaling attempts to provision resources before demand materializes. This paper evaluates a machine-learning-assisted autoscaling approach using a reproducible synthetic workload benchmark. A deterministic generator produces 40,000 five-minute observations with daily and weekly seasonality, autoregressive variation, noise, and burst events. Historical lag and rolling-window features support a 15-minute-ahead request-demand forecast. Persistence, Random Forest, histogram-based gradient boosting, and XGBoost are considered, with chronological train, validation, and test partitions. Across five random seeds, XGBoost obtains mean absolute error of 163.87 requests (standard deviation 1.97), root mean squared error of 208.83 (4.35), and coefficient of determination of 0.7373 (0.0086). In a separate single-seed controller replay, predictive scaling reduces normalized instance-hour cost by approximately 0.29% relative to a reactive threshold policy. However, the fraction of intervals above the 70% CPU proxy rises from 2.70% to 4.39%. These results show that forecast accuracy alone does not guarantee operational improvement. The benchmark is synthetic, cost is normalized, and CPU thresholds are not application-level service-level objectives. Real-trace validation and latency-aware queue simulation remain necessary before production conclusions can be drawn.

Keywords: Cloud Computing, Predictive Autoscaling, Workload Forecasting, Machine Learning, XGBoost, Resource Management.

I. INTRODUCTION

Cloud computing provides on-demand access to shared pools of configurable computing resources. Elasticity allows an application to increase or release capacity as its workload changes, and is a central property of cloud platforms [1], [2]. This flexibility does not remove the resource-management problem: allocating too few instances can increase response time or cause requests to fail, whereas allocating too many instances increases infrastructure expenditure and may leave substantial capacity idle. Autoscaling therefore seeks an operating point that balances service quality, resource utilization, and cost.

Most production autoscaling systems use reactive policies based on observable indicators such as CPU utilization, memory pressure, request rate, or queue length. A controller compares a metric with a configured threshold and then performs a scale-out or scale-in action. Such policies are comparatively simple to implement and explain, but they respond to evidence of load after it has already appeared. Instance provisioning, container scheduling, image retrieval, and application initialization introduce delays between a scaling decision and usable capacity. During a sudden workload increase, this control-loop delay can allow queues to grow and response times to deteriorate before new resources become ready [3], [4].

Predictive autoscaling attempts to address this delay by forecasting future demand and translating the forecast into an earlier resource decision. Forecasting methods range from statistical approaches such as autoregressive models and exponential smoothing to machine-learning models such as Random Forest, gradient boosting, recurrent neural networks, and reinforcement-learning-based controllers. Tree ensembles are useful candidates for tabular workload features because they can represent nonlinear interactions and do not require the extensive sequence modelling associated with some deep-learning architectures. XGBoost, in particular, provides a scalable gradient-boosted tree implementation with regularization and efficient training [5]. However, a lower forecasting error is not by itself proof of lower cost or better application performance. Forecasts must be evaluated inside a controller and under explicit provisioning, cooldown, and service-capacity assumptions.

Prior research has examined cloud autoscaling from several perspectives. Surveys classify rule-based, control-theoretic, queuing-based, and learning-based approaches [3]. Capacity-management research has shown that resource allocation must account for workload dynamics and multi-tier dependencies [6]. More recent systems have explored predictive and reinforcement-learning-based control, including approaches deployed in commercial services [7], while adaptive Kubernetes autoscaling has been studied in operational environments [8]. Work comparing provider autoscaling and queueing/control-based policies also highlights the need to evaluate service-level objectives alongside resource consumption [9]. Public traces such as Google cluster traces provide opportunities to test methods against observed workload behaviour rather than relying exclusively on generated data [10].

The present study asks whether a short-horizon machine-learning forecast can improve the cost and utilization trade-off of a threshold-driven autoscaling policy under a controlled workload. Its contributions are: (1) a deterministic synthetic workload generator with temporal structure and burst events; (2) a chronological forecasting evaluation across multiple random seeds; (3) a comparison of persistence and three tree-based forecasting models; (4) a trace-replay comparison between reactive and predictive threshold controllers; and (5) an explicit analysis of cost, CPU-threshold behaviour, and limitations. The study is intentionally framed as a reproducible simulation benchmark. It does not claim real-provider savings, application-level latency guarantees, or external validity from the synthetic workload.

II. LITERATURE REVIEW

A. Reactive and Predictive Autoscaling

Autoscaling research commonly distinguishes reactive policies from proactive policies. Reactive approaches use current measurements, such as CPU utilization, memory consumption, request rate, or queue length, to trigger scale-out and scale-in actions. Threshold-based policies are comparatively straightforward to implement and interpret, but they may respond only after workload growth has begun. Provisioning and application-startup delays can therefore create a temporary capacity shortfall. Reviews of cloud autoscaling describe threshold rules, queuing-based methods, control-oriented approaches, and learning-based techniques as major families of resource-management strategies [3], [16]. Recent cloud-native research further emphasizes that autoscaling decisions must be evaluated across the application lifecycle and under realistic deployment constraints [17].

B. Workload Forecasting and Machine-Learning Methods

Predictive autoscaling attempts to reduce the reaction delay by estimating future workload and initiating resource provisioning before

demand materializes. Workload forecasting is challenging because cloud applications exhibit temporal variability, burstiness, and heterogeneity. Feng and Ding classify cloud workload prediction research according to application characteristics and explain how forecasts support performance assurance, cost reduction, and energy optimization [18]. Forecasting methods include statistical time-series models and machine-learning techniques. For structured historical features, tree-based ensemble methods can represent nonlinear relationships between calendar patterns, lagged demand, and rolling-window statistics. XGBoost is a regularized gradient-boosting framework designed for scalable tree learning and is therefore a relevant candidate for tabular workload prediction [5].

C. Resource Management, Cost, and Performance Trade-offs

Prior work on large-scale cluster management demonstrates that resource allocation involves balancing utilization, availability, and workload requirements. Google Borg, for example, combines admission control, task packing, over-commitment, and resource sharing to improve cluster utilization [6]. Cloud autoscaling studies also show that reducing resource consumption cannot be considered independently from application performance, because aggressive scale-in or delayed scale-out may increase queueing and service-level risk [4], [16]. Consequently, an autoscaling evaluation should consider both resource or cost measures and service-quality indicators rather than relying on prediction error alone. Public cluster traces provide useful evidence of real workload variability and can support validation beyond synthetic data [10].

D. Research Positioning

The reviewed literature establishes the relevance of workload forecasting and adaptive resource management, but it also highlights the need to evaluate forecasting models inside the scaling policy that consumes their predictions. The present study addresses this evaluation link through a controlled synthetic benchmark: it compares persistence and tree-based forecasting methods, uses a 15-minute prediction horizon, and replays reactive and forecast-driven threshold controllers under common capacity assumptions. Unlike studies based on production traces or deployed autoscalers, the current experiment is intentionally limited to a reproducible simulation. Its normalized cost and CPU-threshold proxy are therefore interpreted as comparative experimental measures, not as evidence of actual cloud-provider savings or application-level SLA guarantees.

III. METHODOLOGY

The research follows an experimental simulation design. Workload generation, feature construction, chronological model evaluation, controller replay, and sensitivity analysis are treated as separate stages. This separation makes it possible to assess forecast quality independently from the resource-management policy that consumes the forecast.

A. Synthetic Workload Generation

A deterministic generator creates 40,000 observations sampled at five-minute intervals, representing approximately 139 days. The request-count series combines daily and weekly periodic components with autoregressive variation, random burst events, and additive noise. Generated demand is bounded to a range of 200–5,000 requests per interval. The purpose of this generator is to create a controlled workload with multiple temporal characteristics so that experiments can be repeated exactly. It is not intended to reproduce the full complexity of a production service, including multi-tenant interference, dependency failures, geographic traffic shifts, or real request-size distributions.

B. Feature Engineering and Prediction Target

The prediction target is request_count at time t+3, equivalent to a 15-minute-ahead forecast at the five-minute sampling interval. Nineteen predictors are constructed from calendar information, historical demand lags, rolling statistics, and demand changes. Lag features use observations at t−1 and earlier; rolling windows summarize the recent history without including future values. The current target value is excluded from the predictor matrix to prevent direct target leakage. Missing rows created by lagging and forward target construction are removed, leaving 39,973 usable observations.

Calendar predictors represent hour-of-day and day-of-week patterns. Lagged request counts use offsets of 1, 2, 3, 6, 12, and 24 observations. Rolling means and standard deviations use windows of 3, 6, 12, and 24 observations. Two demand-delta variables capture recent changes in workload. This feature set represents short-term persistence, local variability, and recurring temporal patterns while remaining interpretable and computationally manageable.

C. Chronological Evaluation and Metrics

The usable time series is divided chronologically into training (70%), validation (15%), and held-out test (15%) partitions. The order is preserved and no random shuffling is applied, reducing the risk that future observations influence model fitting. Hyperparameter selection is performed using the validation segment; final reported test metrics are computed on the held-out segment. Five seeds (11, 22, 33, 44, and 55) are used to regenerate the synthetic series and repeat forecasting evaluation. Results are summarized as mean and standard deviation across these runs.

Forecasting performance is assessed using mean absolute error (MAE), root mean squared error (RMSE), and coefficient of determination (R^2). MAE measures average absolute forecast error in requests; RMSE penalizes larger errors more heavily; R^2 describes the fraction of test-set variance explained relative to the test-set mean. These metrics are complementary and should be interpreted together. A model with a favourable aggregate score may still miss short, high-impact bursts, which is why controller-level metrics are also reported.

$$MAE = (1/n)\,\Sigma|y_i - \hat{y}_i|;\quad RMSE = \sqrt{[(1/n)\,\Sigma(y_i - \hat{y}_i)^2]};\quad R^2 = 1 - \Sigma(y_i - \hat{y}_i)^2 / \Sigma(y_i - \bar{y})^2.$$

D. Forecasting Models

Four forecasting methods are considered. Persistence uses the most recently observed demand as the next-period forecast and provides a simple reference baseline. Random Forest averages predictions from an ensemble of decision trees and can model nonlinear feature interactions. Histogram-based gradient boosting builds trees sequentially using gradient information and histogram binning. XGBoost uses regularized gradient-boosted decision trees and an efficient tree-learning procedure [5]. The single-seed model comparison uses the same chronological data split and feature definitions so that differences are attributable to the learning method rather than to a different evaluation protocol.

E. Reproducibility and Implementation

The model-training stage uses the same input representation and temporal split for each candidate method. The persistence baseline requires no parameter fitting. Random Forest uses a fixed ensemble of 50 trees, maximum depth 10, and minimum leaf size 3. Histogram-based gradient boosting uses 180 boosting iterations, 31 leaves, and a learning rate of 0.08. XGBoost uses 180 boosting rounds, maximum tree depth 6, learning rate 0.05, subsampling and column sampling of 0.85, and the histogram tree method. These settings define the reported benchmark configuration; they should not be interpreted as the result of an exhaustive hyperparameter search.

To reduce optimistic evaluation, feature generation is performed in temporal order and the final test partition is not used for model selection. Forecast errors are calculated only where the future target is available. The multi-seed summary reports dispersion across independently generated workloads, not a confidence interval for a population of cloud applications. Since only five seeds are used, the standard deviation is descriptive and no statistical-significance claim is made.

The experimental design separates three questions: whether a model predicts demand, whether a controller reacts to the forecast in a stable manner, and whether the resulting resource trajectory improves a service objective. The present data support the first question across five seeds and provide a single-seed simulation result for the second. They

do not yet answer the third because end-to-end response times and request failures are not simulated. This separation is important when interpreting the numerical results.

The experiment package contains the generator and training script, pinned Python dependencies, the engineered data, test predictions, metric summaries, controller replay outputs, and threshold-sensitivity results. The random seeds are fixed and written in the experiment configuration. Reproduction requires installing the listed dependencies and executing the main experiment script. Dataset provenance is explicitly labelled synthetic. The project does not include an independently downloaded public production trace, and therefore the reported results must not be described as real-cloud trace validation.

IV. MODELING AND ANALYSIS

A. Predictive Autoscaling Architecture

The evaluated control loop has four stages: workload observation, demand prediction, capacity estimation, and scaling actuation. At each five-minute decision point, the controller receives the available workload history. The predictive policy estimates request demand 15 minutes ahead and converts that forecast into projected CPU utilization under the current fleet size. The reactive policy instead bases its decision on current simulated CPU utilization. Both policies use the same scale-out and scale-in thresholds, minimum and maximum fleet sizes, startup delay, and cooldown settings so that the comparison isolates the effect of using a forecast signal.

The control-flow sequence is: (1) collect recent request-count observations; (2) construct lag and rolling-window features; (3) obtain a forecast from the trained model; (4) estimate projected CPU pressure; (5) compare the pressure signal with the scaling thresholds; and (6) issue a one-instance scale action when the controller is not in cooldown. This design is intentionally simple. It provides a transparent baseline for studying the interaction between prediction and threshold-based resource control, rather than claiming to represent a complete cloud orchestration platform.

Workload history	Lag / rolling features
Forecasting model	15-min demand estimate
Capacity estimator	Projected CPU proxy
Scaling controller	Threshold + cooldown
Instance fleet	Scale out / in

Figure 1. Forecast-driven autoscaling control flow evaluated in the simulation.

B. Resource and Capacity Model

The simulator assumes that one active instance can process 500 requests per five-minute interval at 100% CPU. Given demand d and n active instances, simulated CPU utilization is computed as min(100, 100d/(500n)). The fleet begins with two instances and is constrained to a minimum of two and a maximum of 50. A scaling action changes the desired fleet by one instance. These assumptions translate demand into a CPU proxy, but they do not model heterogeneous virtual-machine performance, memory pressure, network bottlenecks, request-size variation, or downstream service saturation.

C. Scaling Policies and Timing

The reactive controller scales out when current simulated CPU exceeds 70% and scales in when CPU falls below 30%. The predictive controller applies the same thresholds to the projected CPU derived from the 15-minute demand forecast. Both controllers use a 10-minute startup delay and a 15-minute cooldown. The startup delay represents the interval between requesting capacity and making it available to serve workload. Cooldown suppresses repeated actions during transient fluctuations. The simulator bills instances during boot from the interval in which launch occurs, avoiding the undercount that would result if provisioning delay were treated as free capacity.

D. Cost and Operational Metrics

Infrastructure expenditure is represented by a normalized instance-hour cost: one cost unit per instance-hour. Because the unit price is not mapped to a specific cloud provider or instance family, results are comparative simulation units rather than monetary savings. The experiment records total cost, mean billable instances, peak active instances, mean CPU, 95th-percentile CPU, scaling-action count, and the fraction of intervals above 70% or below 35% CPU. The high-CPU fraction is an overload proxy only; it is not equivalent to response-time SLO violations or failed requests.

E. Threshold Sensitivity

A sensitivity experiment varies the predictive scale-out threshold across 65%, 70%, 75%, and 80%, and the scale-in threshold across 20%, 25%, 30%, and 35%. Forecast values and the held-out workload remain fixed while the controller thresholds change. This analysis tests whether the reported cost/performance trade-off is strongly dependent on a particular threshold configuration. It does not identify a universally optimal policy because the objective weights assigned to cost, overload, and underutilization are application-specific.

F. Queueing and Latency Scope

A practical extension should represent arrivals as a time-varying process and service times as a distribution estimated from request-level observations. For a simple first model, an M/M/c queue can provide an analytical reference where arrivals are Poisson, service times are exponentially distributed, and c identical instances serve requests in parallel. However, real application traffic may be bursty and service times heavy-tailed; discrete-event simulation or trace-driven queue replay would therefore be preferable for final evaluation. The model should report queue length, waiting time, completed throughput, timeout count, and latency percentiles.

An operational controller should also distinguish desired capacity from ready capacity. A scale-out request does not immediately increase service rate; new instances contribute only after their startup and health-check periods complete. Scale-in should be guarded by minimum replica constraints and connection-draining time to avoid terminating instances that still hold active requests. These details can materially change the value of a forecast because the prediction horizon should cover the provisioning delay plus the time required for the service to absorb incoming traffic.

A queueing-aware evaluation would require a request-arrival process, per-instance service-time distribution, concurrency model, queue discipline, timeout policy, and readiness dynamics. Those components are not present in the current benchmark. Consequently, no p95 or p99 response-time values, queue lengths, request drops, or application SLO compliance results are claimed. This is an important boundary: CPU utilization is a resource signal, while latency is an end-to-end service outcome influenced by queueing and application architecture [11]. Future implementation should compare a queueing-based controller with the current threshold policies and validate the model against a real trace.

V. RESULTS AND DISCUSSION

A. Forecasting Performance Across Random Seeds

The five-seed evaluation indicates that XGBoost learns useful structure from the generated workload. Across seeds 11, 22, 33, 44, and 55, mean MAE is 163.87 requests with a standard deviation of 1.97. Mean RMSE is 208.83 requests with a standard deviation of 4.35, while mean R^2 is 0.7373 with a standard deviation of 0.0086. The relatively small standard deviations indicate that the forecasting result is stable across the selected generator seeds. This stability applies only to the synthetic

data-generating process and should not be interpreted as evidence of generalization to unrelated cloud applications.

Metric	Mean across 5 seeds	Standard deviation
MAE (requests)	163.87	1.97
RMSE (requests)	208.83	4.35
R^2	0.7373	0.0086

Table I. XGBoost forecasting performance across five synthetic workload seeds.

B. Comparison with Forecasting Baselines

In the single-seed chronological comparison, XGBoost obtains MAE 162.90, RMSE 204.76, and R^2 0.7424. Histogram-based gradient boosting and Random Forest produce competitive results, while persistence is less accurate. The difference between XGBoost and the other tree ensembles is modest, suggesting that feature construction and workload regularity contribute substantially to performance. A wider model study should include seasonal ARIMA or exponential smoothing, LightGBM, recurrent networks, and probabilistic forecasts, with tuning budgets and test windows controlled consistently.

Model	MAE	RMSE	R^2
Persistence	191.97	241.09	0.6429
Random Forest	165.00	207.71	0.7350
Hist. Gradient Boosting	164.37	206.62	0.7377
XGBoost	162.90	204.76	0.7424

Table II. Single-seed held-out forecasting comparison.

C. Reactive and Predictive Controller Comparison

The results should be read as a joint cost-and-risk comparison rather than a ranking based on one metric. The reduction in action count suggests that the predictive signal produces a less frequently changing control trajectory under the selected cooldown. Fewer actions can reduce orchestration overhead and configuration churn, but action count alone does not establish stability or improved user experience. Similarly, the lower mean fleet size is consistent with the small cost reduction, yet the higher 95th-percentile CPU and high-CPU interval fraction indicate less headroom during some periods.

Because the cost difference is less than one percent, small changes in the assumed instance price, startup delay, workload burst timing, or scale thresholds could reverse the comparison. The normalized cost model is useful for controlled comparison but cannot support a currency-denominated savings claim. A production study should use the relevant provider's billing granularity and include instance families, autoscaling minimums, discounts, storage, network transfer, and any charges incurred while resources are starting.

The controller replay uses 5,996 held-out timesteps. Reactive scaling incurs 3,040.25 normalized cost units, while predictive scaling incurs 3,031.33, corresponding to a reduction of approximately 0.29%. Mean billable fleet size changes from 6.085 to 6.067 instances. Predictive scaling also reduces the number of scaling actions from 624 to 196. However, the fraction of intervals above the 70% CPU proxy increases from 2.70% under reactive control to 4.39% under predictive control. The fraction below 35% CPU also increases from 6.55% to 11.61%, indicating a different utilization distribution rather than an unambiguous efficiency improvement.

Measure	Reactive	Predictive
Total normalized cost	3040.25	3031.33
Mean billable instances	6.085	6.067
Peak active instances	9	7
Mean CPU (%)	50.12	49.94
95th percentile CPU (%)	66.81	69.41
CPU >70% intervals	2.70%	4.39%
CPU <35% intervals	6.55%	11.61%
Scaling actions	624	196

Table III. Reactive and predictive controller outcomes on the held-out synthetic workload.

V. RESULTS AND DISCUSSION (CONTINUED)

D. Threshold Sensitivity

Threshold sensitivity confirms that controller configuration materially changes the cost and CPU-threshold trade-off. For example, a 70% scale-out and 35% scale-in configuration reduces normalized cost to 2,858.25 units, but the high-CPU proxy rises to 5.84%. A more conservative 65% scale-out and 30% scale-in configuration records 3,143.58 cost units and a 1.80% high-CPU fraction. At an 80% scale-out threshold, cost falls to 2,763.67 units while the high-CPU fraction rises to 15.38%. These outcomes demonstrate that optimizing cost in isolation can move the system toward a higher overload-proxy rate.

Scale-out / scale-in	Cost units	CPU >70%	CPU <35%	Actions
65% / 30%	3143.58	1.80%	12.68%	260
70% / 25%	3252.17	2.23%	18.73%	72
70% / 30%	3031.33	4.39%	11.61%	196
70% / 35%	2858.25	5.84%	5.54%	320
75% / 30%	2928.08	8.89%	10.36%	156
80% / 30%	2763.67	15.38%	7.71%	84

Table IV. Selected points from the predictive threshold-sensitivity experiment.

E. Interpretation of Findings

The forecasting results and controller results lead to different conclusions. The forecasting experiment supports the use of tree-based models for short-horizon demand estimation on this synthetic workload. XGBoost provides the best single-seed metrics among the compared methods and exhibits low variation across the five seeds. Nevertheless, the improvement over histogram gradient boosting and Random Forest is small. This indicates that model choice should be assessed against complexity, inference cost, retraining requirements, and robustness—not forecast accuracy alone.

At the controller level, the observed cost reduction is only 0.29%, which is too small to support a strong claim of meaningful cloud cost optimization. Predictive control reduces scaling actions, but it also increases the proportion of intervals above the chosen CPU threshold. This trade-off can arise when forecast errors or controller thresholds delay scale-out or retain fewer instances during a demand increase. The result reinforces a key systems principle: a forecasting model must be evaluated as part of the closed-loop resource manager, including actuation delay and service constraints [7]–[9].

F. Limitations and Threats to Validity

External validity also depends on the meaning of the workload signal. A cluster trace may record CPU utilization, task submissions, or container resource usage rather than end-user request arrivals. Such fields cannot be substituted directly for request count without a defensible mapping. Trace preprocessing must preserve timestamps, identify missing intervals, handle machine or task lifecycle events, and avoid mixing training and test periods. Results should be reported

separately by workload type or service where the trace contains heterogeneous jobs.

Further comparisons should include at least one statistical forecaster (for example, seasonal ARIMA or exponential smoothing), a strong gradient-boosting alternative such as LightGBM, and a sequence model such as LSTM or GRU if sufficient training data are available. On the control side, Kubernetes Horizontal Pod Autoscaler, a queueing-based controller, and a published predictive or reinforcement-learning policy should be replayed under identical constraints. The same test traces, provisioning delays, capacity model, and objective metrics must be used to make the comparison meaningful.

There are several limitations. First, the workload is synthetic; its periodicity, burst frequency, and noise structure may not represent real production traces. Second, the multi-seed protocol currently evaluates forecasting robustness, while the reported controller replay is a single-seed experiment. Third, the capacity model assumes homogeneous instances with fixed service capacity. Fourth, the normalized cost model is not a cloud-provider price model and omits discounts, storage, network transfer, reserved capacity, and billing granularity. Fifth, CPU-threshold fractions are not substitutes for latency, throughput, error rate, or request-drop measurements. Sixth, the study does not yet compare against a queueing-theoretic controller, Kubernetes HPA, ARIMA/Prophet, LightGBM, LSTM, or reinforcement-learning autoscalers under a shared evaluation protocol.

Finally, no real public cloud trace has yet been replayed through the complete pipeline. Google cluster traces, Azure workload traces, and Alibaba cluster traces are useful sources for external validation [10], [12], [13], but each has different measurement semantics and sampling intervals. A future study must document the selected trace version, filtering decisions, workload mapping, and licensing conditions. These limitations mean that the current work should be interpreted as a reproducible controlled benchmark, not a production-ready autoscaling system.

VI. CONCLUSION

This paper evaluated machine-learning-assisted predictive autoscaling using a reproducible synthetic workload benchmark. A 15-minute-ahead demand forecaster was trained from calendar, lag, rolling-window, and demand-change features. Across five random seeds, XGBoost achieved mean MAE of 163.87 requests, mean RMSE of 208.83, and mean R^2 of 0.7373. In a separate single-seed simulation, predictive scaling reduced normalized instance-hour cost by approximately 0.29% compared with a reactive threshold controller. However, the CPU-overload proxy increased from 2.70% to 4.39%, showing that a small cost reduction can accompany a less favourable high-utilization profile.

The main conclusion is that forecast quality and autoscaling effectiveness are related but distinct evaluation targets. A predictive controller must account for capacity readiness, workload uncertainty, threshold selection, and application-level service objectives. The present findings are limited to synthetic demand and a simplified CPU/cost model. They do not establish real cloud savings or latency compliance. Future work should replay the method on at least one documented public cloud trace, repeat full controller experiments across multiple seeds, include queueing and service-time distributions, report p50/p95/p99 latency and request drops, and compare against established statistical, machine-learning, Kubernetes, control-theoretic, and reinforcement-learning policies.

VII. REFERENCES

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” NIST Special Publication 800-145, 2011.

[2] M. Armbrust et al., “A View of Cloud Computing,” Communications of the ACM, vol. 53, no. 4, pp. 50–58, 2010, doi: 10.1145/1721654.1721672.

[3] L. Qu et al., “A Survey of Auto-Scaling Techniques in the Cloud Computing,” ACM Computing Surveys, vol. 51, no. 3, 2018, doi: 10.1145/3148149.

[4] A. Gandhi et al., “Autoscale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers,” ACM Transactions on Computer Systems, vol. 30, no. 4, 2012, doi: 10.1145/2382553.2382556.

[5] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in Proc. 22nd ACM SIGKDD, 2016, pp. 785–794, doi: 10.1145/2939672.2939785.

[6] A. Verma et al., “Large-scale cluster management at Google with Borg,” in Proc. EuroSys, 2015, pp. 1–17, doi: 10.1145/2741948.2741964.

[7] Y. Zheng, J. Li, and J. Zhang, “A Meta Reinforcement Learning Approach for Predictive Autoscaling in the Cloud,” in Proc. 28th ACM SIGKDD, 2022, pp. 4290–4299, doi: 10.1145/3534678.3539063.

[8] Z. Zhou et al., “AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes,” in Proc. AAAI, vol. 37, no. 13, 2023, pp. 15621–15629, doi: 10.1609/aaai.v37i13.26852.

[9] “Autoscaling Solutions for Cloud Applications Under Dynamic Workloads,” IEEE Transactions on Services Computing, vol. 17, no. 3, pp. 804–820, 2024, doi: 10.1109/TSC.2024.3354062.

[10] C. Reiss et al., “Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis,” in Proc. ACM Symposium on Cloud Computing, 2012.

[11] R. Buyya et al., “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” Future Generation Computer Systems, vol. 25, no. 6, pp. 599–616, 2009, doi: 10.1016/j未来.2008.12.001.

[12] I. Pintye, J. Kovács, and R. Lovas, “Enhancing Machine Learning-Based Autoscaling for Cloud Resource Orchestration,” Journal of Grid Computing, vol. 22, art. 68, 2024, doi: 10.1007/s10723-024-09783-1.

[13] M. Kaviani, P. Asghari, and M. Sabeti, “A comprehensive survey and taxonomy of workload forecasting techniques in cloud computing,” Computer Networks, 2026, doi: 10.1016/j.comnet.2026.112454.

[14] “Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications,” IEEE Access, vol. 9, pp. 35464–35476, 2021, doi: 10.1109/ACCESS.2021.3061890.

[15] S. J. Taylor and B. Letham, “Forecasting at Scale,” The American Statistician, vol. 72, no. 1, pp. 37–45, 2018, doi: 10.1080/00031305.2017.1380080.

[16] S. Alharthi, A. Alshamsi, A. Alseiari, and A. Alwarafy, “Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions,” Sensors, vol. 24, no. 17, art. 5551, 2024, doi: 10.3390/s24175551.

[17] B. Jeong and Y.-S. Jeong, “Autoscaling Techniques in Cloud-Native Computing: A Comprehensive Survey,” Computer Science Review, vol. 58, art. 100791, 2025, doi: 10.1016/j.cosrev.2025.100791.

[18] B. Feng and Z. Ding, “Application-Oriented Cloud Workload Prediction: A Survey and New Perspectives,” Tsinghua Science and Technology, vol. 30, no. 1, pp. 34–54, 2025, doi: 10.26599/TST.2024.9010024.