International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 5, Issue 2, 241-249, 2024
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V5I2P127

Original Article

Transformer-Based Multi-Signal Predictive Autoscaling for SLA-Aware Resource Management in Kubernetes-Orchestrated Cloud-Native Environments

Nilesh Mutyam
Senior Software Development Engineer, Walmart GTP/PRE, Dallas, TX, USA.

Abstract *- Kubernetes has become the dominant orchestration substrate for cloud-native applications, yet its native autoscaling mechanisms remain primarily reactive, threshold-driven, and limited in their ability to anticipate workload volatility before service-level agreement violations occur. Modern microservice systems exhibit non-linear interactions among request arrival rates, queueing delays, CPU saturation, memory pressure, network variability, pod cold-start latency, and downstream dependency bottlenecks. These characteristics make single-metric autoscaling policies insufficient for latency-sensitive workloads operating under strict service-level objectives. This paper proposes a Transformer-Based Multi-Signal Predictive Autoscaling framework for SLA-aware resource management in Kubernetes-orchestrated cloud-native environments. The proposed framework integrates heterogeneous observability signals, multi-horizon time-series forecasting, uncertainty-aware decision logic, and Kubernetes-native actuation to allocate resources before overload conditions materialize. Unlike conventional Horizontal Pod Autoscaler configurations that respond after resource utilization crosses predefined thresholds, the proposed approach forecasts near-future demand and performance risk using a Transformer encoder architecture designed to learn long-range dependencies, temporal seasonality, burst behavior, and cross-metric interactions. The framework translates predicted workload and latency risk into safe scaling actions through policy constraints that consider replica bounds, cooldown windows, pod readiness delays, cost budgets, and SLA violation probability. The paper develops the conceptual architecture, methodological workflow, evaluation metrics, and analytical discussion necessary for empirical implementation. The study argues that SLA-aware predictive autoscaling should be treated not merely as a forecasting task but as an integrated control problem involving observability quality, model calibration, decision governance, and runtime safety. The proposed model contributes to cloud resource management research by aligning deep temporal learning with Kubernetes operational semantics and by providing a structured pathway toward more reliable, efficient, and self-adaptive cloud-native platforms.*

Keywords *– Kubernetes, Predictive Autoscaling, Transformer, Cloud-Native Systems, Service-Level Agreement, Service-Level Objective, Microservices, Resource Management, Observability, Time-Series Forecasting, Horizontal Pod Autoscaler, AIOps.*

1. Introduction

Cloud-native computing has transformed enterprise software delivery by decomposing applications into independently deployable microservices, packaged as containers, and orchestrated through platforms such as Kubernetes. This architectural transition enables rapid release cycles, elastic scaling, and fine-grained operational control, but it also introduces new challenges in resource management. A single user-facing transaction may traverse dozens of services, each with distinct CPU, memory, I/O, network, and dependency characteristics. As a result, performance degradation is rarely caused by one isolated resource metric. It emerges from interactions among workload arrival patterns, internal queues, downstream services, scheduling delays, and orchestration decisions. Kubernetes inherited key design principles from large-scale cluster management systems, including declarative desired state, automated reconciliation, and controller-driven operations [14]. However, the elasticity mechanisms used in many production clusters still depend heavily on reactive control loops.

The default Kubernetes Horizontal Pod Autoscaler expands or contracts the number of pod replicas based on observed metrics such as CPU utilization, memory utilization, or custom metrics. Although this mechanism is practical and widely adopted, it typically responds after demand has already changed. In latency-critical systems, delayed reaction can cause request queues to accumulate before additional pods become ready. By the time new capacity is scheduled, image-pulled, initialized, warmed, and integrated into load balancing, service-level objective violations may already have occurred. Empirical studies of Kubernetes HPA behavior have shown that metric selection, measurement windows, scaling intervals, and custom metric design significantly influence autoscaling effectiveness [3]. Therefore, autoscaling must be understood as a closed-loop control process with non-trivial delays rather than as a simple resource-threshold mechanism.

The need for more intelligent autoscaling has become stronger as organizations increasingly deploy workloads with bursty traffic, diurnal seasonality, external event spikes, and strict SLA obligations. Classical cloud autoscaling research has long recognized that elasticity must balance competing objectives such as performance, cost, stability, and resource utilization [5]. In Kubernetes environments, this balance becomes more difficult because microservice architectures produce distributed performance dependencies and because containerized workloads can be rapidly rescheduled across heterogeneous infrastructure. Reactive autoscaling may preserve simplicity, but it often underperforms under sharp workload ramps, flash crowds, and latency-sensitive service chains. Conversely, overly aggressive scaling may reduce latency at the cost of overprovisioning, replica oscillation, and wasted infrastructure spend.

Machine learning offers a pathway toward predictive and adaptive scaling by learning future workload behavior from historical and real-time signals. Recent software engineering research has emphasized the expanding role of ML models in adaptive software lifecycle practices, intelligent decision support, and operational optimization [4]. This perspective is directly relevant to cloud-native resource management, where autoscaling decisions must increasingly rely on probabilistic inference rather than static rules. However, applying ML to autoscaling requires more than inserting a forecasting model into the control loop. The model must ingest heterogeneous observability data, handle missing and noisy telemetry, account for pod readiness lag, quantify prediction uncertainty, and generate actions compatible with Kubernetes APIs and operational guardrails.

Transformers are particularly promising for this setting because they can model long-range temporal dependencies and cross-feature interactions through attention mechanisms. The original Transformer architecture demonstrated the power of attention-based sequence modeling by replacing recurrent computation with self-attention, allowing models to learn dependencies across variable-length sequences more effectively [2]. In time-series forecasting, Transformer variants have been used to capture seasonality, trend, multi-horizon dependencies, and exogenous covariates. Informer introduced efficient attention mechanisms for long sequence forecasting, reducing the computational burden associated with full self-attention while preserving predictive capacity over long temporal windows [6]. These properties make Transformer-based forecasting suitable for autoscaling scenarios in which future demand is shaped by both recent bursts and longer historical patterns.

This paper proposes a Transformer-Based Multi-Signal Predictive Autoscaling framework for SLA-aware Kubernetes resource management. The central thesis is that autoscaling should integrate multiple telemetry streams, predict future SLA risk across short time horizons, and convert those predictions into controlled scaling decisions. The proposed framework uses CPU, memory, request throughput, queue depth, latency percentiles, error rate, pod readiness state, network saturation, and dependency health as joint input signals. It forecasts workload and latency-risk trajectories and determines whether scale-out, scale-in, or hold actions should be executed. The design is inspired by decision intelligence principles, which emphasize structured, evidence-driven governance over complex technology lifecycles and architecture-centered management [1]. In this paper, such governance is applied to runtime elasticity: scaling decisions must be accurate, explainable, constrained, and auditable.

The contribution of this paper is fourfold. First, it formulates SLA-aware Kubernetes autoscaling as a multi-signal predictive control problem rather than a single-metric utilization problem. Second, it proposes a Transformer-based methodology for learning temporal and cross-metric dependencies from observability data. Third, it defines a Kubernetes-native architecture that integrates data collection, forecasting, decision policy, actuation, and feedback monitoring. Fourth, it presents a comprehensive evaluation framework covering SLA compliance, prediction accuracy, resource efficiency, stability, cost, and operational safety. The paper is conceptual and methodological in nature; it does not claim production deployment results, but it provides a research-grade design suitable for empirical validation.

2. Background and Related Work

Autoscaling research has evolved from static threshold policies to rule-based, control-theoretic, queueing-based, machine-learning, and reinforcement-learning approaches. Early surveys classify autoscaling systems according to decision mechanisms, resource types, adaptation timing, and optimization objectives [8]. These classifications remain useful for understanding Kubernetes environments, but cloud-native systems introduce additional complexity because workloads are decomposed into many independently scaled microservices. The scaling unit is no longer simply a virtual machine but a pod, deployment, replica set, container resource request, node pool, or service graph segment. This increases the granularity of control while also increasing the probability of local decisions producing global instability.

Kubernetes HPA is the most common horizontal scaling mechanism. It periodically evaluates metrics and adjusts the desired number of replicas for a target workload. Studies of HPA behavior show that CPU-based scaling can be effective for CPU-bound workloads but may perform poorly when application latency depends on queue depth, thread-pool utilization, I/O saturation, or downstream dependency delays [3]. In many microservice systems, CPU utilization can remain moderate while request latency grows due to internal queues or remote service bottlenecks. This motivates the use of custom metrics and multiple signals. Traffic-aware HPA research demonstrates that network and traffic conditions can improve scaling decisions

in edge-oriented Kubernetes environments [11]. Similarly, bi-metric autoscaling studies show that combining CPU utilization with thread-pool or queue-related indicators can better represent service pressure than a single hardware metric [22].

Predictive autoscaling attempts to overcome reactive delay by forecasting future workload or resource demand. Predictive hybrid autoscaling for containerized applications combines workload prediction with horizontal and vertical scaling logic, showing that proactive resource adjustment can address the slow adaptation of purely reactive policies [15]. Related fog and microservice studies also demonstrate that predictive models can be trained to satisfy response-time SLOs under dynamic demand patterns [21]. However, many predictive approaches use relatively limited feature sets or forecasting models that do not fully exploit cross-signal interactions. Kubernetes workloads are often influenced by periodic traffic, deployment events, upstream batch jobs, cache behavior, business calendars, and dependency performance. A forecasting model that treats each metric independently may miss important precursors of SLA violation.

Self-aware and self-adaptive cloud autoscaling research provides a broader theoretical foundation. Self-adaptive autoscaling systems must monitor runtime context, analyze performance states, plan adaptation, and execute changes while considering uncertainty and trade-offs [12]. Reinforcement learning has also been studied for autoscaling because it can optimize sequential decisions under uncertainty, but RL methods require careful reward design, safe exploration, and substantial training data before production use [20]. For Kubernetes operations, the practical challenge is not only selecting an intelligent decision algorithm but also ensuring safe integration with existing control loops, deployment pipelines, observability tools, and organizational policies.

Transformer-based time-series forecasting has advanced rapidly in recent years. Temporal Fusion Transformers combine attention, gating, static covariate encoding, and interpretable multi-horizon prediction, making them useful for complex temporal systems that require both accuracy and interpretability [9]. Autoformer introduced decomposition and autocorrelation mechanisms to improve long-term forecasting by separating trend and seasonal structures [13]. PatchTST further demonstrated that segmenting time series into patches can improve representation learning and reduce attention complexity for long historical windows [16]. These advances are relevant to autoscaling because workload traces often contain multi-scale temporal features: seconds-level bursts, minute-level ramps, daily periodicity, weekly business cycles, and event-driven anomalies.

Microservice benchmarking research further confirms that cloud-native performance is shaped by distributed interactions rather than isolated resource saturation. DeathStarBench, for example, shows that microservices create tail-latency effects and cross-layer performance implications across application, networking, operating system, and hardware layers [17]. This observation is fundamental for SLA-aware autoscaling. A predictive autoscaler must not only forecast incoming request volume but also infer whether the current service graph has enough effective capacity to absorb that volume without violating latency targets. Bottleneck-aware autoscaling work extends this idea by emphasizing that scaling non-bottleneck services can waste resources while failing to solve the actual performance problem [25].

Recent Kubernetes-specific studies have focused on tuning, thresholds, and SLA-adaptive policies. HPA tuning research shows that careful parameter selection can reduce resource waste while maintaining performance under load [18]. SLA-adaptive threshold adjustment work further argues that autoscaling thresholds should be derived from performance objectives rather than selected manually [19]. Industrial adaptive HPA systems also show that business-volume forecasting and pod planning can reduce elastic lag in large-scale production environments [24]. These studies collectively indicate that the next generation of Kubernetes autoscaling should be predictive, multi-signal, SLA-aware, and operationally constrained.

3. Problem Statement

The research problem addressed in this paper is the design of an SLA-aware autoscaling framework that can predict near-future workload and performance risk from heterogeneous observability signals and translate those predictions into safe Kubernetes scaling actions. Let a cloud-native application consist of a set of microservices deployed as Kubernetes workloads. Each service has replica count, resource requests, readiness state, and observed telemetry. The application has one or more SLA or SLO targets, such as p95 latency below 300 ms, error rate below 0.1%, or availability above 99.9%. The autoscaler must decide at each control interval whether to scale out, scale in, or hold each workload.

The difficulty arises from five interconnected issues. First, Kubernetes scaling decisions are delayed by metric collection intervals, controller reconciliation periods, scheduler latency, container startup time, readiness probes, cache warm-up, and load-balancer propagation. A decision made at time t may only affect effective capacity at time t + k. Second, workload and latency are not determined by CPU utilization alone. A pod may experience high request latency due to downstream dependency saturation while CPU remains below the threshold. Third, microservice dependencies propagate local bottlenecks through the call graph. Scaling a front-end service may not improve SLA compliance if the bottleneck is a database adapter, message consumer, or recommendation service. Fourth, aggressive autoscaling can produce oscillation, where replicas

repeatedly expand and contract due to noisy metrics. Fifth, scale-in decisions carry reliability risk because premature removal of capacity can increase latency during recurring bursts.

Traditional HPA policies are therefore insufficient for workloads with bursty demand, strict latency targets, or complex service dependencies. Comparative machine learning studies in software defect prediction highlight that algorithm selection, metric design, and performance comparison significantly influence predictive system reliability [7]. The same principle applies to autoscaling: model choice alone is not enough. The system must define the right prediction targets, choose meaningful features, evaluate multiple baselines, and measure operational consequences. Similarly, comprehensive ML performance comparison research reinforces the importance of evaluating predictive methods across multiple criteria rather than relying on a single accuracy score [10].

The problem can be formally described as follows. Given a multivariate telemetry matrix X over historical window W, where each row represents a time step and each column represents a signal such as CPU utilization, memory pressure, request rate, latency percentile, queue depth, error rate, pod readiness, and dependency health, predict future workload and SLA-risk sequence Y over horizon H. The autoscaling policy then maps predicted sequence Y and current cluster state S into action A, where A may be scale-out, scale-in, vertical recommendation, hold, or alert. The objective is to minimize SLA violations and cost while maintaining stability and avoiding unsafe actions.

4. Proposed Framework / Methodology

The proposed framework, named T-MSPA, consists of six methodological layers: telemetry acquisition, feature engineering, Transformer-based forecasting, SLA-risk estimation, policy-constrained decision making, and Kubernetes-native actuation.

The first layer collects multi-signal observability data. Signals include infrastructure metrics such as CPU utilization, memory working set, network throughput, disk I/O, and container restart count. Application metrics include request rate, p50/p95/p99 latency, error rate, active requests, queue depth, thread-pool saturation, garbage-collection time, and event-loop delay for asynchronous runtimes. Kubernetes state metrics include replica count, desired replicas, ready replicas, pending pods, pod age, restart count, readiness-probe status, node pressure, and scheduling failures. Dependency signals include database latency, message broker lag, cache hit ratio, downstream error rate, and external API latency. The purpose is to capture both demand and effective service capacity.

The second layer transforms raw telemetry into model-ready features. Missing values are imputed using forward-fill or interpolation depending on signal type. Outliers are flagged rather than blindly removed, because spikes may represent meaningful precursors of SLA violation. Signals are resampled to a common time interval, such as 15 or 30 seconds, and aligned with Kubernetes control-loop frequency. Feature transformations include rolling means, exponentially weighted moving averages, rate-of-change indicators, lag features, calendar encodings, burst indicators, and dependency-state flags. Because scaling actions alter metric distributions, the framework includes replica-normalized features such as requests per ready pod and CPU per request. This helps distinguish true workload increase from metric changes caused by redistribution across replicas.

The third layer uses a Transformer encoder for multi-horizon forecasting. The input is a sequence of feature vectors over a look-back window. Positional encodings represent temporal order, while optional calendar embeddings represent hour-of-day, day-of-week, and known business-event periods. The Transformer learns attention weights across historical time steps and across feature dimensions. The output includes predicted request rate, latency percentiles, queue depth, and SLA-risk score for multiple horizons, such as 1, 3, 5, and 10 minutes. Multi-horizon prediction is important because Kubernetes actuation delay varies depending on image availability, node capacity, and readiness time.

The fourth layer estimates SLA violation probability. Rather than scaling solely on predicted traffic, the framework predicts whether the application is likely to violate latency or error-rate objectives. This distinction matters because not all traffic growth causes SLA risk, and not all SLA risk is caused by traffic growth. For example, dependency degradation may increase latency even when request rate is stable. The SLA-risk head is trained as either a classification output or a calibrated probabilistic output. A violation label can be generated from historical data by marking windows where p95 latency exceeds the target or error rate exceeds the threshold. The model can also output uncertainty estimates using ensemble Transformers, Monte Carlo dropout, or quantile forecasting.

The fifth layer converts predictions into scaling decisions. The policy engine combines predicted load, predicted SLA risk, uncertainty, current replica count, minimum and maximum replica bounds, cooldown windows, scale-up and scale-down stabilization rules, pod startup latency, and cost budgets. A scale-out action is triggered when predicted SLA risk exceeds a threshold and expected capacity at the decision horizon is insufficient. A scale-in action is triggered only when predicted demand remains below capacity across a longer safety window and uncertainty is low. Hold actions are used when predictions

are uncertain, when scaling would violate budget constraints, or when bottleneck attribution indicates that adding replicas would not improve performance.

The sixth layer executes scaling through Kubernetes-native mechanisms. The framework may update the scale subresource of a Deployment, write external metrics consumed by HPA, or operate as a custom controller. It should not bypass Kubernetes reconciliation semantics. Instead, it should cooperate with existing controllers by producing desired replica recommendations and recording decision metadata. Every scaling decision should be logged with input signals, forecast values, risk estimates, uncertainty levels, and policy constraints. This audit trail supports debugging, compliance, and operational trust.

5. System Architecture or Conceptual Model

The proposed architecture contains five major components: observability pipeline, forecasting service, decision controller, Kubernetes actuation layer, and feedback repository.

The observability pipeline receives metrics from Prometheus, OpenTelemetry collectors, Kubernetes Metrics Server, service mesh telemetry, and application instrumentation. It stores time-series data in a feature store optimized for sliding-window retrieval. Because autoscaling decisions are sensitive to delayed or missing telemetry, the pipeline includes data-quality checks. It marks stale metrics, detects telemetry gaps, and prevents unsafe scaling decisions when key signals are unavailable.

The forecasting service hosts the Transformer model. It exposes an internal prediction API that receives the current service identifier, recent telemetry window, Kubernetes state, and optional deployment metadata. It returns multi-horizon forecasts and uncertainty estimates. For efficiency, inference can be batched across services, and models can be trained globally with service embeddings or individually for critical workloads. A global model improves generalization when services share traffic patterns, while service-specific fine-tuning captures unique performance characteristics.

The decision controller implements the autoscaling policy. It compares forecasted SLA risk against target objectives and determines the desired replica count. The controller accounts for pod startup delay by scaling to the capacity needed at the future time when replicas become ready. For example, if the model predicts that request rate will exceed current capacity in three minutes and average pod readiness time is two minutes, the controller scales before the violation occurs. The decision controller also performs bottleneck checks. If downstream dependency latency is the dominant risk signal, the controller may avoid scaling the caller service and instead recommend scaling the bottleneck service or raising an alert.

The Kubernetes actuation layer applies the desired state. This may be implemented as a Kubernetes operator with custom resource definitions. A custom resource, such as PredictiveAutoscalingPolicy, can specify service-level objectives, feature groups, replica bounds, forecast horizon, safety thresholds, and fallback behavior. The operator reconciles the desired policy with cluster state and writes scale recommendations. This design aligns with Kubernetes controller patterns and avoids introducing an external automation system that conflicts with native orchestration.

The feedback repository stores decisions and outcomes. After each scaling action, the system records whether SLA risk decreased, whether latency improved, whether resource utilization remained efficient, and whether the action caused oscillation. These records support continuous evaluation and model retraining. They also enable post-incident analysis, allowing operators to determine whether SLA violations resulted from forecast error, slow actuation, insufficient node capacity, dependency bottlenecks, or incorrect policy thresholds.

A critical architectural principle is fallback safety. If the forecasting service fails, telemetry becomes stale, or uncertainty exceeds a predefined threshold, the system should revert to a conservative HPA policy or hold current capacity. Predictive autoscaling should enhance reliability, not introduce a new single point of failure. Therefore, the architecture must support graceful degradation, manual override, and controlled rollout.

6. Evaluation Criteria / Performance Metrics

A rigorous evaluation of T-MSPA should measure predictive accuracy, SLA compliance, resource efficiency, scaling stability, cost, and operational safety. Forecasting metrics alone are insufficient because a model with low mean absolute error may still produce poor scaling decisions if it underestimates bursts or overreacts to noise.

The first category is prediction accuracy. Metrics include mean absolute error, root mean square error, mean absolute percentage error, symmetric mean absolute percentage error, quantile loss, and calibration error for SLA-risk probability. Multi-horizon evaluation should report accuracy separately for short, medium, and long horizons. Short horizons capture immediate scale-out needs, while longer horizons capture trend and seasonality.

The second category is SLA compliance. Metrics include SLA violation rate, total violation duration, p95 and p99 latency, error-rate exceedance, and recovery time after load spikes. SLA violation duration is especially important because cloud users experience sustained degradation more severely than isolated threshold crossings. The framework should also measure how early the autoscaler acts before predicted violations.

The third category is resource efficiency. Metrics include average replica count, CPU utilization, memory utilization, request capacity per replica, overprovisioning ratio, underprovisioning ratio, and idle resource cost. The objective is not simply to minimize replicas but to maintain enough capacity to satisfy the SLA with minimal waste. Cost should be evaluated using cloud pricing models or normalized resource-unit cost.

The fourth category is scaling stability. Metrics include number of scaling events, oscillation frequency, replica churn, scale-in reversals, and cooldown violations. A stable autoscaler should avoid frequent small changes that create operational noise and destabilize application performance. Scale-in decisions should be evaluated separately because unsafe scale-in can create avoidable SLA violations.

The fifth category is operational safety. Metrics include fallback activation rate, decisions made under missing telemetry, prediction uncertainty during actions, and percentage of decisions with complete audit records. For production adoption, an autoscaler must be trustworthy and explainable. Operators should be able to inspect why a scaling action occurred and whether it was driven by traffic growth, latency trend, queue depth, dependency degradation, or uncertainty.

The sixth category is comparative performance against baselines. Baselines should include default CPU-based HPA, custom-metric HPA using request rate, tuned HPA, reactive multi-metric HPA, LSTM-based predictive autoscaling, Prophet-based forecasting, and reinforcement-learning approaches where feasible. Evaluation should use synthetic load profiles, real production traces where available, and benchmark microservices such as social-network or e-commerce workloads. The aim is to determine whether Transformer-based multi-signal forecasting improves SLA compliance and cost efficiency under realistic microservice conditions.

7. Results and Discussion / Analytical Discussion

Because this paper proposes a framework rather than reporting a completed empirical deployment, the results section is presented as an analytical discussion of expected behavior, evaluation interpretation, and research hypotheses. The primary hypothesis is that multi-signal Transformer forecasting will reduce SLA violations compared with reactive HPA under bursty and seasonal workloads. This expectation follows from the fact that predictive models can initiate scale-out before overload becomes visible to reactive thresholds. However, the advantage is likely to be strongest when workload patterns contain learnable temporal structure. For purely random shocks with no observable precursors, predictive autoscaling may still require conservative capacity buffers.

The second hypothesis is that multi-signal inputs will outperform single-metric forecasting. CPU utilization is often a lagging indicator of performance stress. Request rate, queue depth, p95 latency trend, pod readiness, and dependency latency can reveal overload earlier. For example, rising queue depth per ready pod may indicate saturation before CPU crosses a threshold. Similarly, increasing downstream latency may indicate that scaling the caller service is not appropriate. Multi-signal attention can learn these interactions and assign higher importance to signals that historically preceded SLA violations.

The third hypothesis is that Transformer models will outperform simpler forecasting models when workloads contain long-range dependencies, calendar effects, and mixed seasonal-burst behavior. Informer, Autoformer, and PatchTST demonstrate that attention-based models can be adapted to long-sequence forecasting through efficient attention, decomposition, or patching strategies [6]. In autoscaling, long historical context is valuable because many production workloads show recurring daily or weekly patterns. However, Transformers may not always outperform simpler models for small datasets or highly stationary workloads. Therefore, model selection should be empirical, and the system should allow fallback to lighter models when appropriate.

The fourth hypothesis is that uncertainty-aware policy constraints will reduce unsafe scaling. Predictive autoscaling can create risk when forecasts are wrong. Overprediction wastes resources, while underprediction causes SLA violations. Uncertainty estimation allows the decision controller to distinguish confident predictions from ambiguous states. For scale-out, high uncertainty may justify conservative extra capacity if the service is critical. For scale-in, high uncertainty should delay removal of replicas. This asymmetric treatment reflects the operational reality that scale-out cost is often less damaging than customer-facing SLA failure.

The fifth hypothesis is that bottleneck-aware interpretation will improve resource efficiency. In microservice systems, scaling the wrong service can increase cost without reducing latency. If attention or attribution analysis identifies dependency

latency as the dominant risk contributor, the controller can suppress unnecessary scale-out of upstream services. This behavior aligns with bottleneck-aware autoscaling principles and prevents replica inflation across the service graph [25].

The analytical comparison with conventional HPA is instructive. HPA is simple, robust, and easy to operate, but it is reactive. It works well when workload changes slowly and CPU utilization correlates strongly with latency. T-MSPA is more complex but potentially superior when workload changes faster than pod readiness time, when latency depends on multiple signals, or when strict SLAs require proactive capacity. Therefore, the proposed framework should not be viewed as a universal replacement for HPA. It should be applied selectively to critical services where SLA failure cost exceeds the complexity cost of predictive control.

8. Practical Implications

The proposed framework has several practical implications for cloud platform engineering teams. First, observability quality becomes a prerequisite for intelligent autoscaling. Teams must instrument applications beyond CPU and memory. Request latency histograms, queue depth, dependency latency, error rate, and pod readiness metrics should be standardized across services. Without reliable telemetry, predictive models will learn incomplete or misleading relationships.

Second, autoscaling policy should be aligned with SLA definitions. Many organizations define SLAs at the business or API level but configure autoscaling at the infrastructure level. T-MSPA bridges this gap by translating latency and reliability objectives into resource decisions. This supports a more mature operating model in which autoscaling is not merely resource optimization but SLA governance.

Third, platform teams should adopt staged deployment. Predictive autoscaling should initially run in shadow mode, generating recommendations without applying them. Operators can compare predicted actions with actual outcomes and tune risk thresholds. The next stage can apply recommendations only for scale-out while keeping scale-in conservative. Full autonomous scaling should be enabled only after sufficient evidence of safety and benefit.

Fourth, cost management should be built into the policy layer. Predictive scaling may increase average replicas if configured aggressively. However, cost-aware constraints can limit unnecessary overprovisioning by requiring sustained predicted demand before scale-out and sustained low-risk forecasts before scale-in. The framework enables differentiated policies: mission-critical services can prioritize SLA protection, while non-critical workloads can prioritize cost efficiency.

Fifth, the framework supports DevOps and AIOps maturity. By logging forecasts, decisions, and outcomes, it creates an evidence base for continuous improvement. Architecture teams can analyze whether SLA risk is caused by insufficient replicas, poor dependency design, slow startup, cold caches, or inefficient code. This connects runtime operations with architecture-centered lifecycle governance and decision intelligence practices [1].

9. Limitations

The proposed framework has limitations. First, Transformer models require sufficient historical data. New services with little telemetry may not benefit immediately. Transfer learning or global service embeddings can reduce this cold-start problem, but service-specific behavior still requires observation.

Second, predictive accuracy depends on telemetry integrity. Missing metrics, incorrect labels, delayed scraping, and inconsistent instrumentation can degrade performance. A predictive autoscaler may make unsafe decisions if data-quality controls are weak. Therefore, telemetry validation is not optional.

Third, model inference and training introduce overhead. Large Transformer models may be unnecessary for simple workloads. Lightweight models may be preferable for small services, low-traffic applications, or clusters with limited control-plane resources. The architecture should support model-size adaptation.

Fourth, autoscaling cannot solve all performance problems. If latency is caused by database locks, inefficient queries, third-party API failure, or regional network outage, adding pods may not help. The framework must therefore include bottleneck detection and avoid treating every SLA risk as a replica shortage.

Fifth, organizational adoption may be difficult. Predictive autoscaling changes operational responsibility. Platform engineers must trust model-driven recommendations, application teams must expose meaningful metrics, and governance teams must define acceptable risk thresholds. The technical framework must therefore be accompanied by operational processes.

10. Future Research Directions

Future research should empirically evaluate T-MSPA using benchmark microservice applications, real cloud traces, and production-like Kubernetes clusters. Experiments should compare default HPA, tuned HPA, custom-metric HPA, LSTM predictors, Prophet-based predictors, reinforcement-learning controllers, and Transformer-based policies under identical load profiles.

A second research direction is explainable autoscaling. Attention weights, feature attribution, and counterfactual analysis can help operators understand why the model recommended scale-out or scale-in. Explainability is especially important for high-impact systems where resource decisions affect reliability, cost, and compliance.

A third direction is service-graph-aware forecasting. Instead of modeling each service independently, future models can incorporate call graphs, dependency topology, and distributed tracing data. Graph neural networks combined with Transformers may improve bottleneck localization and coordinated scaling.

A fourth direction is joint horizontal and vertical autoscaling. Kubernetes workloads often require both replica adjustment and resource-request tuning. Predictive models could recommend not only the number of replicas but also CPU and memory request changes, especially for workloads with variable per-request resource intensity.

A fifth direction is policy verification. Formal methods and simulation-based testing can verify that autoscaling policies satisfy safety properties such as maximum replica churn, minimum capacity preservation, and bounded cost. This would make predictive autoscaling more acceptable in regulated industries.

11. Conclusion

This paper proposed a Transformer-Based Multi-Signal Predictive Autoscaling framework for SLA-aware resource management in Kubernetes-orchestrated cloud-native environments. The central argument is that modern microservice workloads require autoscaling mechanisms that are predictive, multi-signal, uncertainty-aware, and aligned with service-level objectives. Traditional HPA remains valuable because of its simplicity and Kubernetes-native integration, but its reactive nature limits its effectiveness under bursty traffic, strict latency targets, and distributed bottleneck conditions.

The proposed T-MSPA framework integrates heterogeneous observability signals, Transformer-based multi-horizon forecasting, SLA-risk estimation, policy-constrained decision logic, and Kubernetes-native actuation. It treats autoscaling as an intelligent control problem rather than a threshold-crossing event. By forecasting future demand and performance risk, the framework can provision capacity before SLA violations occur. By incorporating uncertainty, bottleneck awareness, and operational guardrails, it can reduce unsafe scaling and improve trust.

The paper contributes a structured methodology, conceptual architecture, evaluation framework, and analytical foundation for future empirical work. The next stage of research should implement the framework in a Kubernetes testbed, evaluate it against strong baselines, and quantify its effects on SLA compliance, cost, resource utilization, and stability. As cloud-native systems continue to grow in complexity, predictive and SLA-aware autoscaling will become a core capability for reliable, efficient, and self-adaptive digital infrastructure.

References

[1] Gunda, S. K., Yettapu, S. D. R., Bodakunti, S., & Bikki, S. B. (2023). Decision Intelligence Methodology for AI-Driven Agile Software Lifecycle Governance and Architecture-Centered Project Management. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 4(1), 102-108. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P112

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in Advances in Neural Information Processing Systems 30, Long Beach, CA, USA, 2017, pp. 5998–6008. Available: https://papers.nips.cc/paper/7181-attention-is-all-you-need

[3] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," Sensors, vol. 20, no. 16, Art. no. 4621, 2020. https://doi.org/10.3390/s20164621

[4] Gunda, S. K. G. (2023). The Future of Software Development and the Expanding Role of ML Models. International Journal of Emerging Research in Engineering and Technology, 4(2), 126-129. https://doi.org/10.63282/3050-922X.IJERET-V4I2P113

[5] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," Journal of Grid Computing, vol. 12, no. 4, pp. 559–592, 2014. https://doi.org/10.1007/s10723-014-9314-7

[6] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting," in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 12, 2021, pp. 11106–11115. https://doi.org/10.1609/aaai.v35i12.17325

[7] Shim, S., Dhokariya, A., Doshi, D., Upadhye, S., Patwari, V., & Park, J. Y. (2023). Predictive auto-scaler for Kubernetes cloud. In 2023 IEEE 17th Annual Systems Conference (SysCon) (pp. 1–8). IEEE. https://doi.org/10.1109/SysCon53073.2023.10131106

[8] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling Web Applications in Clouds: A Taxonomy and Survey," ACM Computing Surveys, vol. 51, no. 4, Art. no. 73, pp. 1–33, 2018. https://doi.org/10.1145/3148149

[9] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, "Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting," International Journal of Forecasting, vol. 37, no. 4, pp. 1748–1764, 2021. https://doi.org/10.1016/j.ijforecast.2021.03.012

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

[11] L. H. Phuc, L.-A. Phan, and T. Kim, "Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure," IEEE Access, vol. 10, pp. 18966–18977, 2022. https://doi.org/10.1109/ACCESS.2022.3150867

[12] T. Chen, R. Bahsoon, and X. Yao, "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems," ACM Computing Surveys, vol. 51, no. 3, Art. no. 61, pp. 1–40, 2018. https://doi.org/10.1145/3190507

[13] H. Wu, J. Xu, J. Wang, and M. Long, "Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting," in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 22419–22430. Available: https://arxiv.org/abs/2106.13008

[14] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," ACM Queue, vol. 14, no. 1, pp. 70–93, 2016. https://doi.org/10.1145/2898442.2898444

[15] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive Hybrid Autoscaling for Containerized Applications," IEEE Access, vol. 10, pp. 109768–109778, 2022. https://doi.org/10.1109/ACCESS.2022.3214985

[16] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, "A Time Series Is Worth 64 Words: Long-Term Forecasting with Transformers," in Proceedings of the International Conference on Learning Representations, 2023. Available: https://arxiv.org/abs/2211.14730

[17] Y. Gan et al., "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 2019, pp. 3–18. https://doi.org/10.1145/3297858.3304013

[18] D. R. Augustyn, Ł. Wyciślik, and M. Sojka, "Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments," Applied Sciences, vol. 14, no. 2, Art. no. 646, 2024. https://doi.org/10.3390/app14020646

[19] O. Pozdniakova, D. Mažeika, and A. Cholomskis, "SLA-Adaptive Threshold Adjustment for a Kubernetes Horizontal Pod Autoscaler," Electronics, vol. 13, no. 7, Art. no. 1242, pp. 1–28, 2024. https://doi.org/10.3390/electronics13071242

[20] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, "Reinforcement Learning-Based Application Autoscaling in the Cloud: A Survey," Engineering Applications of Artificial Intelligence, vol. 102, Art. no. 104288, 2021. https://doi.org/10.1016/j.engappai.2021.104288

[21] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive Autoscaling of Microservices Hosted in Fog Microdata Center," IEEE Systems Journal, vol. 15, no. 1, pp. 1275–1286, 2021. https://doi.org/10.1109/JSYST.2020.2997518

[22] C. Zhu, B. Han, and Y. Zhao, "A Bi-Metric Autoscaling Approach for N-Tier Web Applications on Kubernetes," Frontiers of Computer Science, vol. 16, no. 3, 2022. https://doi.org/10.1007/s11704-021-0118-1

[23] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in Proceedings of the European Conference on Computer Systems, Bordeaux, France, 2015, pp. 1–17. https://doi.org/10.1145/2741948.2741964

[24] Z. Zhou, C. Zhang, L. Ma, J. Gu, H. Qian, Q. Wen, L. Sun, P. Li, and Z. Tang, "AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes," arXiv:2303.03640, 2023. Available: https://arxiv.org/abs/2303.03640

[25] S. Xie, J. Wang, B. Li, Z. Zhang, D. Li, and P. C. K. Hung, "PBScaler: A Bottleneck-Aware Autoscaling Framework for Microservice-Based Applications," arXiv:2303.14620, 2023. Available: https://arxiv.org/abs/2303.14620