

An auto-scaling approach for serverless environments based on a multi-expert consensus mechanism

Journal of Ambient Intelligence and

Smart Environments

1–26

© The Author(s) 2026

Article reuse guidelines:

sagepub.com/journals-permissions

DOI: 10.1177/18761364261459585

journals.sagepub.com/home/ais



Mobina Kashaniyan¹, Mehrdad Ashtiani¹  and Amirhossein Ghassemi¹

Abstract

Serverless computing offers automatic resource management and pay-per-use execution, but autoscaling remains difficult due to cold-start latency, inter-function dependencies, and highly dynamic workloads. Many existing approaches scale functions independently or rely on a single predictor, which can reduce robustness and cost efficiency. We present a dependency-aware autoscaling framework that unifies bottleneck identification, short-horizon demand forecasting, and cost-aware control in an end-to-end pipeline. We model applications as directed dependency graphs and prioritize high-impact functions using degree centrality. For these bottlenecks, near-term demand is predicted using lightweight supervised models, whose outputs are fused via a performance-weighted probabilistic ensemble inspired by Bayesian model averaging to improve stability under workload variability. The controller also accounts for cold starts and filters candidate actions through a cost-comparison mechanism to balance latency and operational efficiency. Experiments on real workload traces show improved prediction accuracy and more stable scaling decisions than representative baselines; supervised forecasting also consistently outperforms unsupervised clustering for generating autoscaling actions. The primary contribution is a practical system-level design that integrates dependency analysis, ensemble-based prediction, and cost-aware decision-making for robust serverless autoscaling.

Keywords

serverless computing, autoscaling, dependency graphs, ensemble learning, cold-start awareness, Bayesian-inspired weighting, resource optimization

1 Introduction

Serverless computing allows developers to deploy event-driven functions without managing infrastructure, offering automatic provisioning and pay-per-use billing. Its fine-grained autoscaling adjusts function instances to workload variations, but accurate control remains challenging due to bursty demand, cold-start latency, and inter-function dependencies (Jonas et al., 2019; Tari et al., 2024; Tournaire et al., 2023; Wen et al., 2023). Unlike VM- or container-based systems, serverless applications often execute as chains of dependent function invocations. Congestion at one function can propagate along execution paths, increasing end-to-end latency and cost (Bibal Benifa and Dejeu, 2019). Autoscalers that treat functions independently may overlook structurally critical bottlenecks, while aggressive scale-down can increase cold starts and degrade performance. To address these challenges, we propose a dependency-aware autoscaling framework that integrates structural bottleneck identification, multi-model demand forecasting, cold-start awareness, and cost-comparison control within a unified pipeline. Workflows are represented as directed dependency graphs, and bottleneck functions are prioritized using degree centrality. Near-term demand for these functions is predicted using lightweight supervised models (MLP, LSTM, and CNN), and predictions are combined via a performance-weighted probabilistic ensemble inspired by

¹ School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Corresponding author:

Mehrdad Ashtiani, School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran, Iran, Postal Code: 16846-13114.

Email: m_ashtiani@iust.ac.ir

Bayesian model averaging to reduce model-specific bias and improve stability. The controller also incorporates cold-start awareness and a cost-comparison step to balance latency and operational efficiency. We evaluate the framework on real workload traces and compare it with standard autoscaling baselines and learning-based approaches. Results show improved prediction accuracy and more stable scaling behavior, and a unified comparison indicates that supervised forecasting substantially outperforms unsupervised clustering for autoscaling decision generation.

The main contributions of this work are as follows:

1. **Dependency-aware autoscaling pipeline:** An end-to-end controller that couples dependency analysis, forecasting, and cost-aware actuation rather than scaling each function independently.
2. **Graph-based bottleneck identification:** A directed dependency graph with degree-centrality ranking to construct a watch set of high-impact functions for targeted monitoring and control.
3. **Lightweight multi-model forecasting:** Demand for bottleneck functions is predicted using three lightweight neural network models: multilayer perceptron (MLP), long short-term memory (LSTM), and convolutional neural network (CNN). These models capture complementary workload patterns and remain computationally suitable for online control.
4. **Bayesian-inspired consensus decision-making:** A probabilistic ensemble combines predictions from multiple models using validation-based weighting to improve stability and reduce model-specific bias.
5. **Cold-start aware and cost-comparison scaling:** The scaling policy takes into account cold-start considerations and aims to control costs effectively.

The remainder of the paper is organized as follows. Section 2 provides essential background. Section 3 reviews related work and identifies open gaps. Section 4 describes the proposed framework and implementation details. Section 5 reports experimental results and evaluation. Section 6 concludes the paper, and Section 7 presents future research directions.

2 Background knowledge

This section briefly introduces the concepts necessary to understand the proposed autoscaling framework. We summarize the serverless execution model, describe dependency modeling through graph representation, and outline the learning paradigms used for demand prediction.

2.1 Serverless execution model

Serverless computing is commonly implemented through function-as-a-service (FaaS), where the cloud provider manages provisioning, scaling, and billing of function instances (Manner, 2023; Shafiei et al., 2022). As shown in Figure 1, responsibility progressively shifts from the customer to the provider across deployment models, from on-premises to infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), FaaS, and software-as-a-service (SaaS). In FaaS, developers primarily manage application logic and data, while the provider controls the runtime, operating system, virtualization layer, and hardware. Serverless functions are typically stateless and event-driven, and are billed according to execution time and allocated memory (Song et al., 2024; Xu et al., 2023). A distinguishing feature is scale-to-zero capability, which releases resources during idle periods. While this improves cost efficiency, reactivating a function may introduce cold-start latency due to runtime initialization (Wen et al., 2022). These characteristics make autoscaling both fine-grained and highly dynamic, requiring decisions that balance responsiveness, latency, and cost under rapidly changing workloads.

2.2 Dependency modeling

Serverless applications commonly consist of multiple interacting functions connected through invocation relationships. A single request may traverse several functions, forming execution paths whose combined behavior determines end-to-end latency and resource consumption. To capture this structure, we represent the application as a directed dependency graph, where nodes correspond to functions and edges represent invocation dependencies (Madsen, 2022). This abstraction enables analysis of structural influence across the workflow. Graph analysis can identify functions that occupy critical positions within execution paths and are therefore more likely to accumulate workload pressure or propagate latency (Li et al., 2025). Prioritizing these structurally influential functions provides a principled basis for targeted autoscaling decisions.

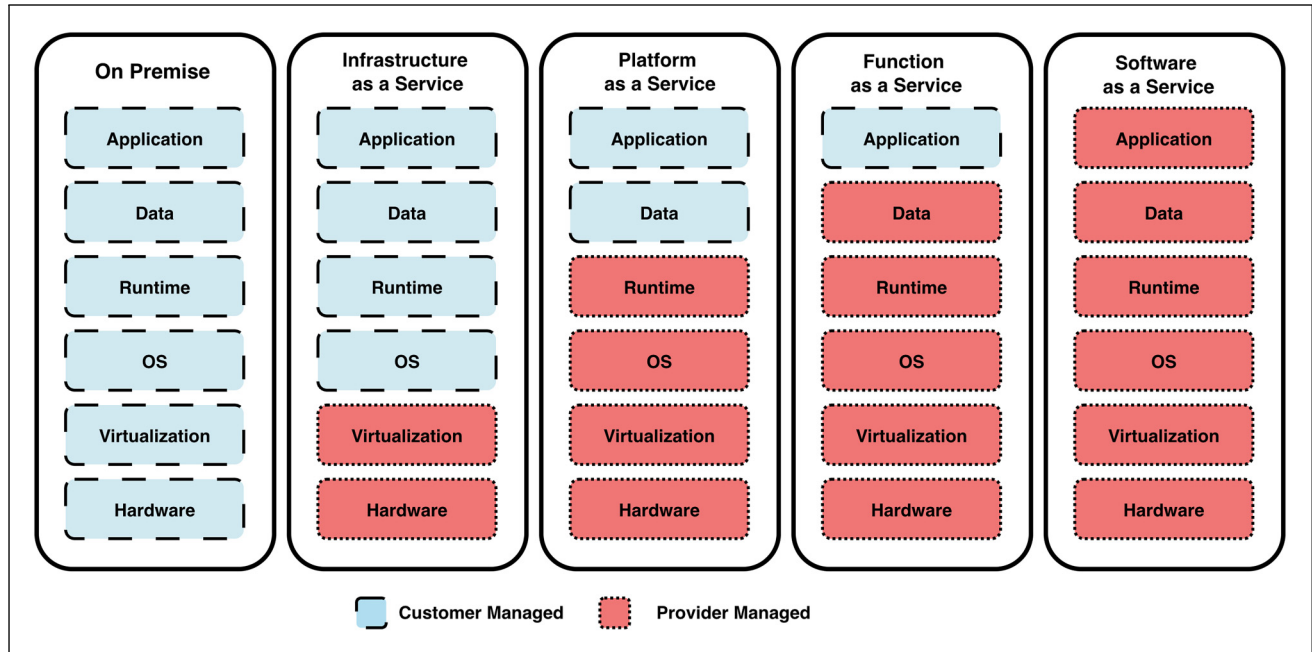


Figure 1. Responsibility distribution across cloud service models. Blue indicates customer-managed components and red indicates provider-managed components. In FaaS, users manage primarily application logic, while the provider controls the underlying infrastructure (Mampage et al., 2022).

2.3 Learning-based demand prediction

Learning-based approaches are widely used in autoscaling to estimate near-term resource demand under dynamic workloads. In this work, supervised models generate actionable scaling decisions, while unsupervised techniques are evaluated only as comparative baselines.

2.3.1 Unsupervised learning (baselines). Unsupervised methods identify patterns in data without labeled outputs. Principal component analysis (PCA) reduces dimensionality by projecting data onto directions that capture maximum variance (Abdi and Williams, 2010; Denton et al., 2021; Ma~kiewicz and Ratajczak, 1993). Clustering techniques, such as k-means, partition observations into similarity-based groups (Ahmed et al., 2020; Berahmand et al., 2025; Ling and Weiling, 2025; Rokach and Maimon, 2006), while self-organizing maps (SOM) provide topology-preserving clustering and visualization (Abdelsamea et al., 2014; Florida et al., n.d; Qu et al., 2021). We also use t-SNE for the visualization of high-dimensional workload patterns (Chang, 2025; Mittal et al., 2024). Although these techniques reveal structural patterns in resource traces, they do not directly produce stable real-time scaling decisions. Therefore, they are used only to evaluate whether pattern discovery alone can approximate autoscaling actions.

2.3.2 Supervised learning (decision models). Supervised learning models learn mappings from observed resource metrics to future demand or scaling actions. MLP captures nonlinear relationships through fully connected layers (Popescu and Balas, n.d; Rana et al., 2018; Singh and Banerjee, 2019). LSTM networks model temporal dependencies in sequential workload data (Landi et al., 2021; Lu and Salem, 2017; Pulver and Lyu, 2017; Siarni-Namini et al., 2019; Smagulova and James, 2019; Yao et al., n.d; Zhao et al., 2020). CNN extract local patterns from sliding windows of time-series inputs (Alzubaidi et al., 2021; Elngar et al., 2021; Purwono et al., 2022; Sahu and Dash, 2021). In the proposed framework, these complementary models serve as candidate predictors, and their outputs are later combined using Bayesian-inspired model averaging to improve robustness.

2.4 Auto-scaling overview

Autoscaling dynamically adjusts compute resources to maintain performance objectives while controlling operational cost. The control granularity varies across platforms, including virtual machines and containers in instance-based clouds (Chen et al., 2019; Hu et al., 2025; Roy et al., n.d; Wang et al., 2024), service replicas in microservices architectures

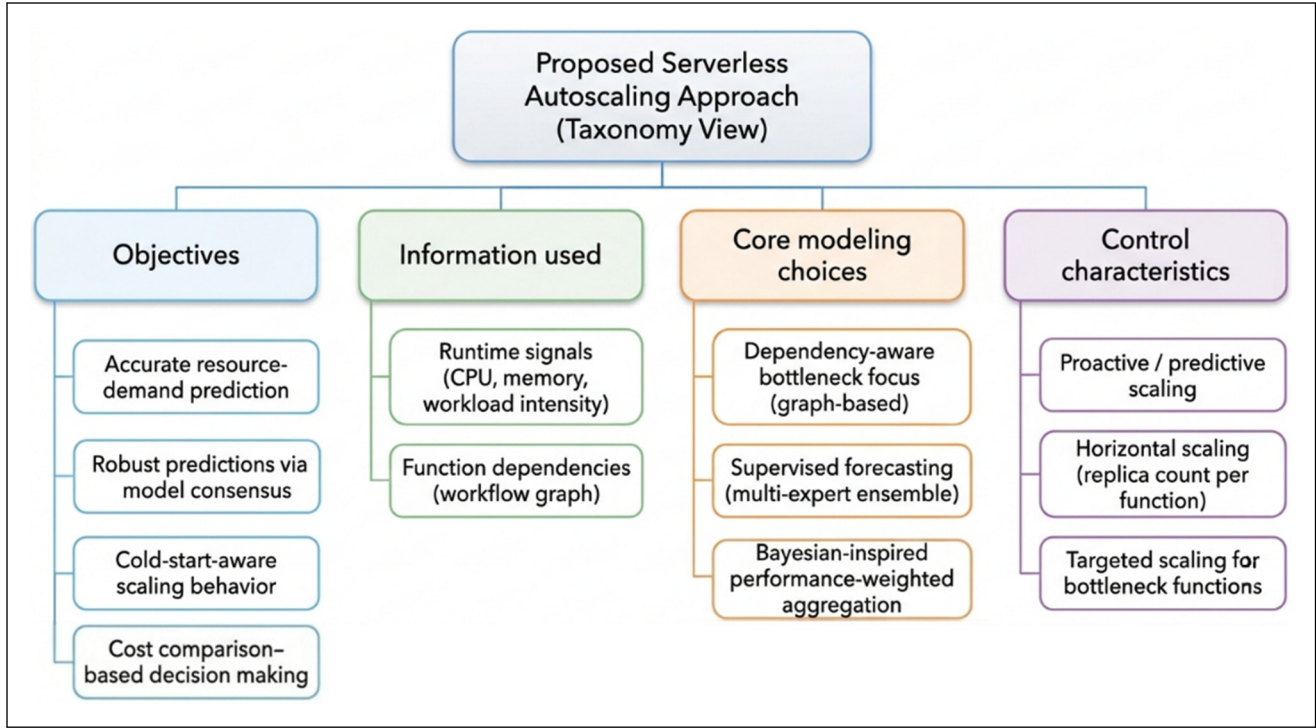


Figure 2. Taxonomy view of the proposed serverless autoscaling approach.

(Nunes et al., 2024; Semerikov et al., 2024), and individual function instances in serverless systems (Mampage et al., 2023). Figure 2 presents a taxonomy view of the proposed framework across four dimensions: objectives, information sources, core modeling choices, and control characteristics. The framework aims to achieve accurate demand prediction, robust consensus-based decision-making, cold-start-aware behavior, and cost comparison-based scaling. Scaling decisions are guided by runtime signals and workflow dependencies; demand is forecast using supervised models aggregated via Bayesian-inspired consensus; and candidate scaling actions are evaluated through a cost-comparison component before execution.

Autoscaling approaches are commonly categorized by how decisions are generated. Threshold-based approaches trigger scaling when monitored metrics exceed predefined limits (Dashtbani and Tahvildari, 2025). Control-theoretic methods regulate system outputs toward target values using feedback mechanisms (Al-Dulaimy et al., 2022). Learning-based approaches use supervised prediction or reinforcement learning to select scaling actions (Ginarsa and Santoso, 2025; Robino et al., 2025; Santos et al., 2025; Valkenborg et al., 2023). Queueing-based models estimate delay and required capacity from traffic characteristics (Jafarnejad Ghomi et al., 2019; Pandey, 2025). Time-series forecasting methods predict short-horizon demand from historical observations (Ding et al., 2025; Fog et al., 2025). Although these categories differ in modeling assumptions and responsiveness, serverless environments introduce additional constraints, including cold-start latency and inter-function dependencies. These characteristics motivate autoscaling designs that integrate predictive modeling with structural awareness, as reflected in the proposed framework.

2.5 Bayesian-inspired model averaging

Ensemble methods combine predictions from multiple models to improve robustness and generalization performance (Wang et al., 2023). Unlike methods that select a single best model, Bayesian Model Averaging (BMA) accounts for model uncertainty by forming a probabilistic weighted ensemble (Bazrafshan et al., 2022; Zhang et al., 2023). BMA has been successfully applied in regression, classification, and time-series forecasting tasks. In this study, BMA is used to produce consensus resource-demand predictions. Let M_i denote candidate models and D denote the observed data. The posterior probability of the model M_i given data D is computed using Bayes' theorem as defined in Equation (1):

$$P(M_i|D) = \frac{P(D|M_i) P(M_i)}{P(D)} \quad (1)$$

Where $P(D|M_i)$ is the likelihood of the data under the model M_i and $P(M_i)$ is the prior probability of the model (Fragoso et al., 2018; Khan, 2023). The marginal likelihood is described in Equation (2):

$$P(D) = \sum_i P(D|M_i)P(M_i) \quad (2)$$

The BMA prediction is obtained as the posterior-weighted average of individual model predictions, as shown in Equation (3):

$$y_{BMA} = \sum_i P(M_i|D)y_i \quad (3)$$

where y_i is the prediction of the model M_i . For decision-level aggregation, the weighted consensus decision is in Equation (4):

$$D_{BMA} = \sum_i w_i D_i \quad (4)$$

where D_i denotes the decision from the model M_i , and weights satisfy $\sum_i w_i = 1$. The normalized weights are defined as in Equation (5):

$$w_i = \frac{P(M_i|D)}{\sum_j P(M_j|D)} \quad (5)$$

Where the probability $P(M_i|D)$ is the posterior probability of model M_i given the observed data D and the denominator represents the sum of posterior probabilities across all candidate models. BMA integrates model predictions according to their posterior probabilities, reducing reliance on any single predictor and improving robustness under workload variability (Arbel et al., 2023; Hinne et al., 2020). In this study, posterior model weights are approximated using validation performance as a proxy for model evidence and are normalized via a softmax transformation to obtain probabilistic weights.

3 Related work

This section situates our work within serverless autoscaling research. We organize prior studies by decision scope (placement vs. scaling), decision mechanism (rules, control, or learning), and use of structural information. We then highlight recurring limitations that motivate our framework.

3.1 Placement vs. scaling decisions

Several studies focus on execution placement, determining where function invocations execute. Weighted scheduling and multi-objective optimization improve throughput, latency, and energy efficiency by assigning requests to suitable nodes, particularly in heterogeneous or edge environments (Aslanpour et al., 2024; Chitsaz et al., 2023). Platform-level designs such as OpenWhisk, OpenFaaS, and Fission introduce schedulers, admission control, and monitoring to enhance dispatch efficiency and reduce local queuing (Govindarajan and Tienne, 2023; Han et al., 2012; Koperek and Funika, 2012). These efforts primarily optimize where requests execute rather than how much capacity should be allocated to each function. They do not explicitly address scaling decisions when requests traverse multiple dependent functions along a call graph. Our work complements placement strategies by directing scaling and pre-warming toward structurally critical functions that bound end-to-end performance.

3.2 Cold-start mitigation

Cold starts remain a major source of latency under bursty demand and scale-to-zero behavior. Stochastic optimization has been used to determine pre-spawning policies that balance responsiveness and energy cost (Anselmi et al., 2025). However, many pre-warming approaches treat functions independently and do not exploit the dependency structure to prioritize which functions should be warmed first. In contrast, our framework leverages a dependency graph to identify bottleneck functions and target scaling and pre-warming accordingly.

3.3 Rule-based and control-theoretic approaches

Rule-based autoscaling remains widely adopted due to its simplicity and low overhead. Dynamic thresholds, escalation policies, mixed-resource rules, and deadline-aware schemes improve responsiveness relative to static policies (Lorido-Botran et al., 2013; Mampage et al., 2021; Maurer et al., 2011). Nevertheless, such approaches are typically reactive, sensitive

to parameter tuning, and applied independently to each function. Control-theoretic and queueing-based methods regulate delay, utilization, or queue length under bursty workloads (Dutreilh et al., 2010; Gambi and Toffetti, 2012; Lim et al., 2010; Roy et al., n.d). While these approaches offer stability guarantees, they generally operate at the service tier and rarely incorporate invocation-graph structure when allocating capacity. As a result, scaling decisions may not prioritize functions that dominate end-to-end latency.

3.4 Learning-based autoscaling

Learning-based methods enable proactive scaling through demand forecasting or policy learning. Reinforcement learning has been applied to elasticity control in systems such as Knative under non-stationary workloads (Rao et al., 2009; Schuler et al., 2020; Zhang et al., 2022). Supervised predictors, including LSTM, CNN, and MLP models, forecast workload or resource utilization to reduce service violations under dynamic traces. Additional approaches employ profit-oriented provisioning or Markov models to estimate required instance counts (Anonymous, 2016; Kumar et al., 2018; Salah et al., 2016; Villela et al., 2004). Although these methods improve proactive decision-making, many rely on a single predictor and primarily use local signals. This can reduce robustness under workload drift, noise, and short-lived bursts. Hybrid controllers combining rules, prediction, and control have also been proposed (Almeida et al., 2002; Fang et al., 2012; Golshani and Ashtiani, 2021; Khatua et al., 2010; Nadjaran Toosi et al., 2019; Sfakianakis et al., 2022; Zhao et al., 2021), yet explicit dependency-aware prioritization and structured cold-start reasoning are often absent. Recent work explores multi-model forecasting to improve prediction accuracy. For example, Taha et al. (2024) propose an MLP-LSTM hybrid, Javeed et al. (2025) combines SVM, random forest, and deep neural networks, and investigates ensemble learning with hyperparameter tuning. Approaches such as (Ouhame et al., 2021) and (Sabyasachi et al., 2024) integrate CNN-LSTM architectures and vector autoregression for enhanced time-series prediction. While these models report improved forecasting accuracy, they typically focus on prediction alone and do not integrate scaling control, dependency-aware bottleneck targeting, or probabilistic consensus for model fusion.

3.5 Recurring limitations

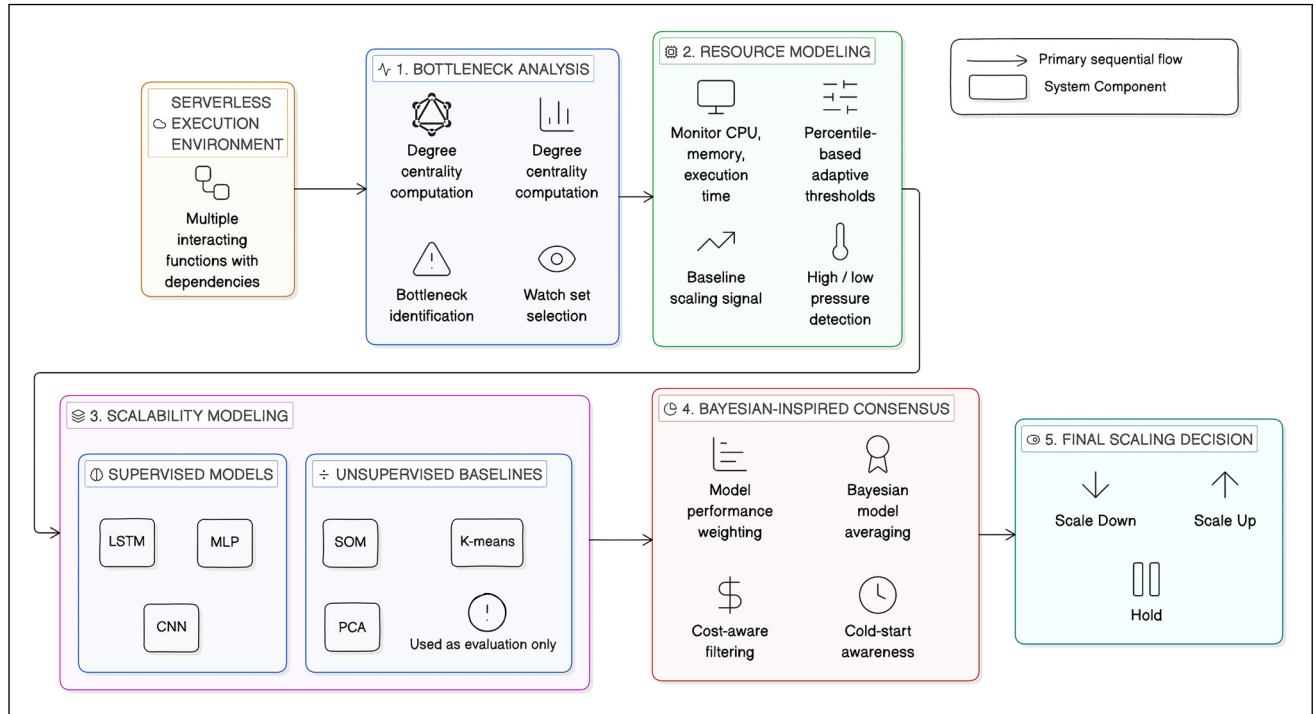
Across the literature, two limitations recur. First, inter-function dependencies are frequently ignored or handled indirectly, even though a small subset of functions along critical invocation paths can dominate tail latency and cost. Second, scaling decisions often rely on a single predictive model, reducing robustness under workload uncertainty and drift. The proposed framework addresses these gaps by modeling function interactions as a directed call graph, identifying bottleneck functions using graph-based measures, forecasting near-term demand with complementary lightweight models MLP, CNN, and LSTM, and combining predictions through Bayesian-inspired model averaging. This produces a dependency-aware, model-robust autoscaler that focuses scaling actions on functions with the greatest end-to-end impact. Table 1 summarizes the differences between our approach and representative learning-based methods.

4 The proposed approach

This section presents the proposed dependency-aware autoscaling framework for serverless applications, illustrated in Figure 3. The framework integrates structural analysis, runtime monitoring, predictive modeling, and consensus-based decision logic into a unified control pipeline. Specifically, dependency analysis constructs a watch set of high-impact functions; resource modeling generates runtime signals and action labels; supervised models forecast next-step demand for the watch set; the consensus module fuses predictions; and the cost-comparison module filters the proposed action before execution. The pipeline begins with Bottleneck Analysis, where the function dependency graph is analyzed using degree centrality to identify structurally critical functions. A watch set is constructed to prioritize monitoring and scaling decisions for these bottleneck candidates. In the Resource Modeling stage, runtime signals including CPU utilization, memory usage, and execution time are continuously monitored. Percentile-based adaptive thresholds generate baseline scaling indicators and detect high or low resource pressure. Next, scalability modeling predicts near-term resource demand for the watch set. Supervised learning models, LSTM, MLP, and CNN, provide complementary forecasts, while unsupervised techniques, SOM, K-means, and PCA, are included solely for comparative evaluation. Predictions from supervised models are aggregated in the consensus module. Model performance weighting and Bayesian-inspired model averaging produce a robust consensus estimate. At this stage, cost comparison and cold-start awareness are incorporated to stabilize scaling decisions. Finally, the Scaling Decision module issues one of three actions: scale up, scale down, or hold, based on the consensus output. By combining dependency-aware prioritization, multi-model forecasting, probabilistic aggregation, and cost comparison, the framework enables targeted and stable horizontal scaling under dynamic workloads.

Table 1. Comparison with related learning-based methods.

Method	Architecture summary	Inputs	Preprocessing	Cold start handling	Limitations
MLP-LSTM (Taha et al., 2024)	Hybrid MLP + LSTM	CPU, memory, bandwidth	Normalization; windowing	No	Designed for VNF/SFC; no cold-start handling
ML Models (Javeed et al., 2025)	SVM / RF / DNN	Big data metrics	Cleaning, normalization, and feature selection	No	Not serverless-aware; no cold-start handling
ML with Hyperparameter Tuning (Vaghasia et al., 2025)	Ensemble ML on the cloud	Large-scale datasets	Normalization; hyperparameter tuning	No	Focuses on prediction only; no runtime autoscaling
CNN-LSTM (Ouham et al., 2021)	CNN + LSTM	CPU, memory	Stationarity check, windowing	No	No autoscaling; single dataset
DCNN LSTM (Sabyasachi et al., 2024)	Deep CNN + LSTM	Resource-usage time series	Normalization; fixed budget	No	No autoscaling, Lower predictive accuracy
Our Ensemble Method	MLP + CNN + LSTM with consensus	CPU + platform signals	Normalization; sliding windows	Yes	Dynamic Autoscaling, better prediction accuracy

**Figure 3.** Dependency-aware multi-expert autoscaling framework architecture, illustrating bottleneck analysis, resource monitoring, predictive modeling, consensus, and final scaling decision.

4.1 Bottleneck analysis component

Serverless workflows typically consist of multiple interacting functions connected through invocation dependencies. Congestion in a single function can propagate along critical execution paths, significantly affecting end-to-end latency and cost. Rather than detecting overload reactively based solely on runtime metrics, the proposed framework proactively identifies structurally influential functions using dependency analysis. The application is represented as a directed dependency graph, where nodes correspond to functions and edges represent invocation relationships. Degree centrality is computed for each node to quantify its structural influence. This measure is selected for its low computational overhead and suitability for dynamic environments that require frequent updates. Functions with high degree centrality interact with many other components and are therefore more likely to amplify workload pressure across execution paths. The framework constructs a watch set consisting of the highest-ranked nodes. Subsequent monitoring, forecasting, and scaling decisions are prioritized for this reduced set of structurally critical functions.

4.2 Degree centrality on the dependency graph

Bottleneck identification begins by constructing a weighted, directed dependency graph from the dependency dataset D with columns $\{ \text{class}, \text{file}, \text{fanin}, \text{fanout} \}$. Each distinct class and file is mapped to a unique node identifier, enabling consistent representation of dependency relationships. A directed edge is created from a class node c_i to a file node f_j whenever the class depends on the file. The edge weight captures dependency intensity using fan-in and fan-out metrics as defined in Equation (6):

$$w(c_i, f_j) = \text{fanin}(c_i) + \text{fanout}(c_i) \quad (6)$$

If multiple records exist for the same class–file pair, weights are accumulated to reflect repeated interactions. The resulting graph is defined as $G = (V, E, w)$, where the vertex set V is partitioned into class nodes V_C , and file nodes V_F , and edges restricted to $E \subseteq V_C \times V_F$. This directed bipartite structure prevents artificial dependencies between nodes of the same type. For each class node $c_i \in V_C$, weighted degree centrality is defined as in Equation (7).

$$C_D(c_i) = \sum_{f_j \in N(c_i)} w(c_i, f_j) \quad (7)$$

where $N(c_i) \subseteq V_F$ denotes the set of file nodes connected to c_i . To enable ranking and comparison, centrality scores are normalized as in Equation (8).

$$C_D^{\text{norm}}(c_i) = \frac{C_D(c_i)}{\max_{c_k \in V_C} C_D(c_k)} \quad (8)$$

The bottleneck score is defined directly as the normalized degree centrality as defined in Equation (9).

$$\text{Score}(c_i) = C_D^{\text{norm}}(c_i) \quad (9)$$

Nodes are ranked in descending order of $\text{Score}(c_i)$, and the top- k nodes form the bottleneck watch set. Algorithm 1 summarizes this procedure and outputs the structurally high-impact functions that are subsequently prioritized for monitoring, forecasting, and scaling.

4.3 Resource modeling component

After identifying bottleneck functions, the resource modeling component derives adaptive scaling thresholds from historical runtime behavior to support stable control decisions. Let D_{hist} denote the historical time series of runtime metrics for a selected bottleneck. Rather than relying on fixed thresholds, the controller computes percentile-based bounds that reflect empirical workload behavior. The component continuously monitors execution time ($\text{Runs_ms} \in \mathbb{R}^+$), average CPU utilization ($\text{AvgCPU} \in [0, 100]$), per-core CPU utilization ($\text{CPUCore}[j] \in [0, 100]$), memory usage ($\text{MemUsed} \in \mathbb{R}^+$), and available memory ($\text{MemAvail} \in \mathbb{R}^+$). Let $P_p(X)$ denote the p th percentile of metric X computed from D_{hist} . Upper thresholds, such as P_{95} capture high-pressure conditions that may require scaling up, while intermediate and lower thresholds, such as P_{70} and P_{05} , define safe operating region. The use of distinct upper and lower bounds introduces hysteresis and reduces oscillatory behavior. Scaling decisions follow a prioritized three-way rule. A high-pressure state is triggered if any monitored metric exceeds its upper threshold. Logical OR is used to ensure responsiveness to emerging bottlenecks. If no high-pressure condition is detected, a low-pressure state is evaluated. Scale-down occurs only when all monitored metrics

Algorithm 1. Bottleneck Identification via Weighted Degree Centrality

```

1: Input: A directed weighted dependency graph  $G = (V_C \cup V_F, E, w)$ , where  $V_C$  is
   the set of class nodes,  $V_F$  is the set of file nodes,  $E \subseteq V_C \times V_F$  represents
   dependency edges, and  $w(c, f)$  denotes the weight of dependency  $(c, f)$ ; and an
   integer  $k$ , the number of bottlenecks to return.
2: Output: Top- $k$  bottleneck class nodes ranked by bottleneck score.
3: Compute weighted degree centrality
4: for each class node  $c \in V_C$  do
5:    $C_D(c) \leftarrow 0$ 
6:   for each edge  $(c, f) \in E$  do
7:      $C_D(c) \leftarrow C_D(c) + w(c, f)$ 
8:   end for
9: end for
10: Normalize degree centrality
11:  $C_{max} \leftarrow \max_{c \in V_C} C_D(c)$ 
12: for each class node  $c \in V_C$  do
13:   if  $C_{max} > 0$  then
14:      $C_D^{norm}(c) \leftarrow \frac{C_D(c)}{C_{max}}$ 
15:   else
16:      $C_D^{norm}(c) \leftarrow 0$ 
17:   end if
18: end for
19: Rank bottlenecks
20: Sort class nodes  $c \in V_C$  in descending order of  $C_D^{norm}(c)$ 
21: return top- $k$  class nodes

```

fall within safe bounds. The Logical AND operator is used to prevent premature downscaling. If neither condition is satisfied, the controller maintains the current resource allocation. Algorithm 2 summarizes the complete threshold derivation and decision process.

4.4 Scalability component

The scalability component produces short-horizon demand predictions that enable proactive scaling decisions. To evaluate learning paradigms under a consistent decision framework, both unsupervised and supervised methods are implemented and compared. However, only supervised models are used to drive the final autoscaling decisions. Unsupervised methods identify latent structure in normalized runtime metrics without access to scaling labels. We consider three clustering strategies: K-Means, PCA followed by K-Means, and SOMs. Let X denote the normalized feature matrix constructed from runtime metrics. Each method produces cluster assignments $z \in \{1, \dots, k\}$. Because cluster indices do not directly correspond to scaling actions, a post-processing step maps each cluster to one of the three scaling decisions $\{-1, 0, 1\}$ (scale-down, hold, scale-up). The mapping is defined using majority voting on labeled training data: each cluster is assigned the most frequent scaling label among its members. This mapping remains fixed during evaluation to ensure comparability with supervised methods. Algorithm 3 details the full procedure.

In contrast, supervised learning directly models the mapping from runtime metrics to scaling actions. The problem is formulated as a three-class classification task with inputs derived from CPU utilization, memory usage, and execution time features. We evaluate three lightweight neural architectures selected for complementary modeling capabilities while maintaining low inference overhead: The MLP model captures nonlinear relationships among aggregated features, the CNN extracts localized patterns within sliding windows of resource metrics, and the LSTM captures temporal dependencies across short look-back windows. Time-series samples are constructed using a fixed look-back window ℓ . Features are standardized prior to training. Each model outputs class probabilities through a softmax layer. Model configurations are intentionally compact to preserve online feasibility. The MLP uses one hidden layer with 64 units and ReLU activation, followed by dropout (0.2) and a softmax output. The LSTM model uses a single LSTM layer with 64 units, dropout (0.2), and softmax output; to limit inference latency, inputs are provided as a short sequence window. The CNN applies a one-dimensional convolution with 32 filters of width three, global max pooling, a dense layer with 64 units, and a softmax output. All models are trained using the Adam optimizer with categorical cross-entropy loss and early stopping to prevent overfitting. Time-series cross-validation is applied to preserve temporal structure. Algorithm 4 summarizes the supervised

Algorithm 2. Resource Modeling & Scaling Decision Using Historical Thresholds

```

1: Input: historical metrics  $D_{hist}$ ; current metrics  $x$  with fields Run_ms,
   AvgCPU, CPUCore[1..4], MemUsed, MemAvail
2: Output: decision  $\in \{-1, 0, 1\}$  (1 = SCALE_UP, 0 = HOLD, -1 = SCALE_DOWN) and
   thresholds
3: Step 1: Derive thresholds from history
4:  $\tau_{cpu}^{hi} \leftarrow P_{95}(D_{hist}.AvgCPU)$ 
5:  $\tau_{cpu}^{lo} \leftarrow P_{70}(D_{hist}.AvgCPU)$ 
6:  $\tau_{mem}^{hi} \leftarrow P_{95}(D_{hist}.MemUsed)$ 
7:  $\tau_{mem}^{lo} \leftarrow P_{70}(D_{hist}.MemUsed)$ 
8:  $\tau_{avail}^{lo} \leftarrow P_{05}(D_{hist}.MemAvail)$ 
9:  $\tau_{run}^{hi} \leftarrow P_{95}(D_{hist}.Run\_ms)$ 
10: thresholds  $\leftarrow \{\tau_{cpu}^{hi}, \tau_{cpu}^{lo}, \tau_{mem}^{hi}, \tau_{mem}^{lo}, \tau_{avail}^{lo}, \tau_{run}^{hi}\}$ 
11: Step 2: Aggregate current metrics
12: peakCPU  $\leftarrow \max_{j \in \{1,2,3,4\}} x.CPUCore[j]$ 
13: Step 3: Evaluate high-pressure condition
14: cpuHigh  $\leftarrow (x.AvgCPU > \tau_{cpu}^{hi}) \vee (peakCPU > \tau_{cpu}^{hi})$ 
15: memHigh  $\leftarrow (x.MemUsed > \tau_{mem}^{hi})$ 
16: availLow  $\leftarrow (x.MemAvail < \tau_{avail}^{lo})$ 
17: runHigh  $\leftarrow (x.Run\_ms > \tau_{run}^{hi})$ 
18: pressurehi  $\leftarrow cpuHigh \vee memHigh \vee availLow \vee runHigh$ 
19: if pressurehi then
20:   return (1, thresholds) ▷ SCALE_UP
21: end if
22: Step 4: Evaluate low-pressure condition
23: cpuLow  $\leftarrow (x.AvgCPU < \tau_{cpu}^{lo}) \wedge (peakCPU < \tau_{cpu}^{lo})$ 
24: memLow  $\leftarrow (x.MemUsed < \tau_{mem}^{lo})$ 
25: availSafe  $\leftarrow (x.MemAvail > \tau_{avail}^{lo})$ 
26: pressurelo  $\leftarrow cpuLow \wedge memLow \wedge availSafe$ 
27: if pressurelo then
28:   return (-1, thresholds) ▷ SCALE_DOWN
29: else
30:   return (0, thresholds) ▷ HOLD
31: end if

```

training procedure. Performance is evaluated with accuracy, precision, recall, and F1-score, as well as mean absolute error (MAE), mean squared error (MSE), averaged across time-series cross-validation folds.

4.5 Consensus component

Relying on a single predictor can make scaling decisions sensitive to workload drift, noise, or transient bursts. The consensus component addresses this by combining predictions from the MLP, CNN, and LSTM using a performance-weighted probabilistic ensemble inspired by Bayesian model averaging principles. Let $M = \{M_{MLP}, M_{LSTM}, M_{CNN}\}$ denote the trained models. Each model produces class probability estimates P_i on the test data. Model performance is evaluated on validation data to obtain accuracy scores A_i that reflect recent predictive reliability. Model weights are computed using a softmax transformation with temperature parameter τ as defined in Equation (10).

$$w_i = \frac{\exp\left(\frac{A_i}{\tau}\right)}{\sum_j \exp\left(\frac{A_j}{\tau}\right)} \quad (10)$$

This formulation ensures higher-performing models receive greater influence, all weights remain positive and $\sum_i w_i = 1$. The ensemble probability distribution is then computed as in Equation (11).

$$P_{ens} = \sum_i w_i P_i \quad (11)$$

Algorithm 3. Unsupervised Learning-Based Scalability Decision (K-Means, PCA + K-Means, SOM)

```

1: Input: dataset  $D$ ; number of clusters  $k$  (default 3); SOM grid shape  $(h, w)$ ;
   PCA components  $p$ ; random seed  $seed$ 
2: Output: cluster labels  $z_{km}, z_{pca}, z_{som}$ 
3: Step 1: Build feature matrix
4:  $X \leftarrow \text{ExtractFeatures}(D)$  ▷  $X$  uses
   {Run_ms, AvgCPU, CPUCore[1..4], MemUsed, MemAvail}
5: Step 2: Normalize features
6:  $X_s \leftarrow \text{MinMaxScale}(X, [0, 1])$ 
7: Step 3: K-Means clustering on normalized features
8:  $km \leftarrow \text{KMeans}(k, seed)$ 
9:  $km.fit(X_s)$ 
10:  $z_{km} \leftarrow km.predict(X_s)$ 
11: Step 4: SOM-based clustering
12:  $som \leftarrow \text{SOM}(grid=(h, w), dim=cols(X_s), \sigma=1.0, \eta=0.5, seed)$ 
13:  $som.init\_weights(X_s)$ 
14:  $som.train(X_s, iters=100)$ 
15:  $W \leftarrow [som.winner(x) \text{ for each } x \in X_s]$   $W$  are winner grid coordinates
16:  $z_{som} \leftarrow \text{KMeans}(k, seed).fit\_predict(W)$ 
17: Step 5: PCA + K-Means
18:  $pca \leftarrow \text{PCA}(p, seed)$ 
19:  $X_p \leftarrow pca.fit\_transform(X_s)$ 
20:  $km_p \leftarrow \text{KMeans}(k, seed)$ 
21:  $z_{pca} \leftarrow km_p.fit\_predict(X_p)$ 
22: return  $z_{km}, z_{pca}, z_{som}$ 

```

Final scaling decisions are obtained via Equation (12).

$$\hat{y}_{ens} = \arg \max_c P_{ens}(c) \quad (12)$$

This performance-weighted probabilistic aggregation reduces dependence on any single model while preserving complementary predictive information. Algorithm 5 details the complete consensus procedure.

4.6 Cost comparison component

The cost comparison component ensures that consensus-driven scaling actions remain economically justified by estimating their projected operational impact before execution. While overprovisioning increases direct resource expenditure, underprovisioning increases latency and may lead to SLA violations; therefore, scaling decisions must balance cost efficiency and performance reliability. Operational cost at time t is modeled as the sum of three components: base resource cost, waste cost, and overload cost. The base cost represents pay-per-use charges proportional to the number of allocated replicas. Waste cost captures overprovisioning when allocated capacity exceeds actual utilization, reflecting underutilized resources. Overload cost models underprovisioning penalties when demand exceeds capacity, and this is amplified when SLA violations occur. In particular, SLA violations are detected through latency thresholds, and the overload penalty is increased accordingly. This mechanism implicitly incorporates cold-start effects, since scale-down decisions that increase predicted latency beyond SLA thresholds raise the overload cost, discouraging aggressive deprovisioning. Algorithm 6 formalizes the computation of these cost terms and derives the projected total cost.

Given a consensus scaling decision, the controller proposes a candidate replica configuration subject to replica bounds. The projected cost of the candidate configuration is then estimated using predicted next-step utilization and latency. Algorithm 7 performs cost-aware acceptance or rejection using a tolerance margin δ . If the candidate cost is sufficiently lower than the current cost, the scaling action is accepted; if it exceeds the allowable tolerance, the action is rejected, and the system holds the current allocation. This comparison-based filtering prevents economically unjustified scaling, particularly scale-down actions that could trigger cold-start latency or SLA degradation. To ensure runtime feasibility, the overall framework remains computationally lightweight. Degree centrality is computed in linear time with respect to the number of dependency edges and can be updated incrementally. The supervised models are intentionally shallow, and inference requires only a small number of matrix operations per control interval. Retraining is performed periodically using a sliding window of recent observations rather than at every step, limiting overhead while preserving adaptability.

Algorithm 4. Supervised Learning-Based Scalability Decision

```

1: Input: dataset  $D$ ; labels  $y$ ; look-back window  $\ell$ ; number of folds  $k$ ;
   early-stopping patience  $p$ 
2: Output: trained models  $M_{mlp}$ ,  $M_{lstm}$ ,  $M_{cnn}$ ; performance summary  $R$ 
3: Step 1: Feature preprocessing
4:  $X \leftarrow \text{ExtractFeatures}(D)$ 
5:  $\text{scaler} \leftarrow \text{StandardScaler}().\text{fit}(X)$ 
6:  $X_s \leftarrow \text{scaler.transform}(X)$ 
7: Step 2: Create time-series samples
8:  $(X_{seq}, y_{seq}) \leftarrow \text{CreateLaggedDataset}(X_s, y, \ell)$ 
9:  $y_{oh} \leftarrow \text{OneHotEncode}(y_{seq})$ 
10: Step 3: Time-series cross-validation
11:  $CV \leftarrow \text{TimeSeriesSplit}(k)$ 
12:  $R \leftarrow \emptyset$ 
13: for each  $(train, val) \in CV$  do
14:    $X_{tr}, X_{val} \leftarrow X_{seq}[train], X_{seq}[val]$ 
15:    $y_{tr}, y_{val} \leftarrow y_{oh}[train], y_{oh}[val]$ 
16:   MLP model
17:    $X_{tr}^f \leftarrow \text{Flatten}(X_{tr})$ 
18:    $X_{val}^f \leftarrow \text{Flatten}(X_{val})$ 
19:    $M_{mlp} \leftarrow \text{BuildMLP}()$ 
20:    $\hat{y}_{mlp} \leftarrow \text{TrainEval}(M_{mlp}, X_{tr}^f, y_{tr}, X_{val}^f, y_{val}, p)$ 
21:    $R \leftarrow R \cup \text{Metrics}(\hat{y}_{mlp}, y_{val}, \text{MLP})$ 
22:   LSTM model
23:    $M_{lstm} \leftarrow \text{BuildLSTM}()$ 
24:    $\hat{y}_{lstm} \leftarrow \text{TrainEval}(M_{lstm}, X_{tr}, y_{tr}, X_{val}, y_{val}, p)$ 
25:    $R \leftarrow R \cup \text{Metrics}(\hat{y}_{lstm}, y_{val}, \text{LSTM})$ 
26:   CNN model
27:    $M_{cnn} \leftarrow \text{BuildCNN}()$ 
28:    $\hat{y}_{cnn} \leftarrow \text{TrainEval}(M_{cnn}, X_{tr}, y_{tr}, X_{val}, y_{val}, p)$ 
29:    $R \leftarrow R \cup \text{Metrics}(\hat{y}_{cnn}, y_{val}, \text{CNN})$ 
30: end for
31:  $R \leftarrow \text{AggregateByModel}(R)$ 
32: return  $M_{mlp}$ ,  $M_{lstm}$ ,  $M_{cnn}$ ,  $R$ 

```

Algorithm 5. Consensus Component via Performance-Weighted Ensemble

```

1: Input: trained models  $M=\{M_{MLP}, M_{LSTM}, M_{CNN}\}$ ; validation data  $(X_{val}, y_{val})$ ; test
   data  $X_{test}$ ; temperature  $\tau$ 
2: Output: ensemble predictions  $\hat{y}_{ens}$ ; model weights  $w$ ; ensemble probabilities
    $P_{ens}$ 
3: Step 1: Evaluate model performance
4: for each model  $M_i \in M$  do
5:    $A_i \leftarrow \text{Evaluate}(M_i, X_{val}, y_{val})$ 
6:    $P_i \leftarrow \text{Predict}(M_i, X_{test})$ 
7: end for
8: Step 2: Compute model weights
9:  $w_i \leftarrow \exp(A_i/\tau)$  for all  $i$ 
10:  $w \leftarrow w / \sum_i w_i$ 
11: Step 3: Aggregate predictions
12:  $P_{ens} \leftarrow \sum_i w_i \cdot P_i$ 
13: Step 4: Generate final decisions
14: for each test sample  $k$  do
15:    $\hat{y}_{ens}[k] \leftarrow \text{argmax}_c P_{ens}[k, c]$ 
16: end for
17: return  $\hat{y}_{ens}$ ,  $w$ ,  $P_{ens}$ 

```

Algorithm 6. Cost Component: Cost Calculation

```

1: Input: current replicas  $R_t$ ; actual utilization  $U_t$ ; predicted utilization  $\hat{A}_t$ ; actual latency  $L_t$ ; SLA threshold  $L_{SLA}$ ; unit cost  $C_{base}$ ; multipliers  $C_{waste}$ ,  $C_{overload}$ ; maximum replicas  $R_{max}$ 
2: Output:  $C_{total}(t)$ ;  $C_{base}(t)$ ;  $C_{waste}(t)$ ;  $C_{overload}(t)$ ;  $cost\_savings(t)$ 
3: Step 1: Base resource cost
4:  $C_{base}(t) \leftarrow R_t \cdot C_{base}$ 
5: Step 2: Waste cost (over-provisioning)
6:  $W_t \leftarrow \max(0, \hat{A}_t - U_t)$ 
7:  $C_{waste}(t) \leftarrow R_t \cdot C_{base} \cdot W_t \cdot C_{waste}$ 
8: Step 3: Overload cost (under-provisioning)
9:  $O_t \leftarrow \max(0, U_t - \hat{A}_t)$ 
10:  $P_t \leftarrow [L_t > L_{SLA}]$   $\triangleright P_t=1$  if SLA violated, else 0
11:  $C_{overload}(t) \leftarrow R_t \cdot C_{base} \cdot O_t \cdot C_{overload} \cdot (1 + P_t)$ 
12: Step 4: Total cost and savings (vs. baseline)
13:  $C_{total}(t) \leftarrow C_{base}(t) + C_{waste}(t) + C_{overload}(t)$ 
14:  $C_{baseline}(t) \leftarrow R_{max} \cdot C_{base}$ 
15:  $cost\_savings(t) \leftarrow C_{baseline}(t) - C_{total}(t)$ 
16: return  $C_{total}(t)$ ,  $C_{base}(t)$ ,  $C_{waste}(t)$ ,  $C_{overload}(t)$ ,  $cost\_savings(t)$ 

```

Algorithm 7. Cost-Aware Scaling Decision

```

1: Input: consensus decision  $\hat{y}_{cons} \in \{SCALE\_DOWN, HOLD, SCALE\_UP\}$ ; current replicas  $R_t$ ; current cost  $C_{total}(t)$ ; predicted next-step utilization  $\hat{A}_{t+1}$ ; predicted next-step latency  $\hat{L}_{t+1}$ ; cost tolerance  $\delta$ ; replica bounds  $R_{min}$ ,  $R_{max}$ 
2: Output: next replicas  $R_{t+1}$ ; scaling action  $action$ 
3: Step 1: Propose candidate replicas from consensus
4: if  $\hat{y}_{cons} = SCALE\_UP$  then
5:    $R_{cand} \leftarrow R_t + 1$ 
6: else if  $\hat{y}_{cons} = SCALE\_DOWN$  then
7:    $R_{cand} \leftarrow R_t - 1$ 
8: else
9:    $R_{cand} \leftarrow R_t$ 
10: end if
11:  $R_{cand} \leftarrow \min(R_{max}, \max(R_{min}, R_{cand}))$   $\triangleright$  Clamp to bounds
12: Step 2: Estimate candidate cost
13:  $C_{cand} \leftarrow EstimateCost(R_{cand}, \hat{A}_{t+1}, \hat{L}_{t+1})$ 
14: Step 3: Cost-aware acceptance/rejection
15: if  $C_{cand} \wedge (1 - \delta) C_{total}(t)$  then
16:    $R_{t+1} \leftarrow R_{cand}$ 
17:    $action \leftarrow \hat{y}_{cons}$ 
18: else if  $C_{cand} > (1 + \delta) C_{total}(t)$  then
19:    $R_{t+1} \leftarrow R_t$ 
20:    $action \leftarrow HOLD$ 
21: else
22:    $R_{t+1} \leftarrow R_{cand}$ 
23:    $action \leftarrow \hat{y}_{cons}$ 
24: end if
25: return  $R_{t+1}$ ,  $action$ 

```

5 Evaluations

The proposed framework is evaluated to assess prediction accuracy, decision stability, and cost impact under dynamic workload conditions. Section 5.1 describes the experimental environment. Section 5.2 details the datasets used for dependency modeling and resource-consumption forecasting. Subsequent sections analyze bottleneck identification, threshold-based scaling behavior, unsupervised baselines, supervised prediction performance, consensus aggregation, and cost-aware decision outcomes.

Table 2. Overview of the dependency analysis dataset.

File	Name of the source file
Class	Name of the class
Fan-in	Number of incoming dependencies
Fan-out	Number of outgoing dependencies
Total-con	fan-in+fan-out

Table 3. Sample records from the dependency analysis dataset.

File	Class	Fan-in	Fan-out	Total-con
\\DATASET\\abfiles\\r0\\c17\\after\\ActiveFiltersPanelTest.java	mtgdeckbuilder.frontend.ActiveFiltersPanelTest	0	4	4
\\DATASET\\abfiles\\r0\\c17\\after\\TagViewerTest.java	mtgdeckbuilder.frontend.TagViewerTest	1	6	7
\\DATASET\\abfiles\\r0\\c17\\after\\TagTopicTest.java	mtgdeckbuilder.frontend.topics.TagTopicTest	0	4	4
\\DATASET\\abfiles\\r0\\c17\\after\\TagAddPanel.java	mtgdeckbuilder.frontend.TagAddPanel\$Anonymous1	0	1	1
\\DATASET\\abfiles\\r0\\c17\\after\\TagTopic.java	mtgdeckbuilder.frontend.topics.TagTopic	12	1	13
\\DATASET\\abfiles\\r0\\c17\\after\\NewFilterPanelTest.java	mtgdeckbuilder.frontend.NewFilterPanelTest	0	4	4
\\DATASET\\abfiles\\r0\\c17\\after\\TagAddPanel.java	mtgdeckbuilder.frontend.TagAddPanel	3	4	7
\\DATASET\\abfiles\\r0\\c17\\after\\CardTaggingPanelTest.java	mtgdeckbuilder.frontend.CardTaggingPanelTest	1	8	9

5.1 Environment and implementation

The framework is implemented in Python. TensorFlow and Keras are used to train the supervised learning models, and Pandas supports preprocessing and feature construction. Experiments are conducted in the Google Colab environment.

The implementation follows the modular architecture described in Section 4, including dependency analysis, percentile-based thresholding, supervised forecasting, performance-weighted ensemble aggregation, and cost-aware decision filtering.

5.2 Dataset preparation

Two datasets are used in this study: one for dependency-driven bottleneck identification and one for resource-consumption forecasting. To approximate structural coupling among serverless functions, we use a publicly available software dependency dataset containing code-level structural metrics from a large software project. Although this dataset is not collected from a deployed serverless platform, it serves as a reproducible and structured proxy for modeling dependency relationships. In the absence of publicly available serverless invocation-graph traces, this dataset enables controlled evaluation of structurally informed prioritization strategies while maintaining transparency and reproducibility independent of proprietary platform data. The dataset contains 509,426 records and 53 attributes, where each record corresponds to a file or class component. From this dataset, we extract the file identifier, class identifier, fan-in (number of incoming dependencies), fan-out (number of outgoing dependencies), and total connections (fan-in + fan-out). These attributes are used to construct a weighted directed bipartite dependency graph, where classes and files form distinct node sets and dependency relations define directed edges. Degree centrality is then computed to identify structurally influential components. Table 2 summarizes the extracted attributes used in the analysis.

For workload modeling, resource consumption data are obtained from real hardware performance traces collected using OpenHardwareMonitor (Madsen, 2022). The dataset consists of time-series measurements of CPU utilization, per-core CPU usage, memory utilization, and execution time. These traces serve as workload inputs for demand forecasting and autoscaling evaluation. Before analysis, both datasets undergo cleaning and normalization. The dependency dataset is used exclusively to compute centrality scores and identify structurally high-impact nodes. To evaluate bottleneck-aware scaling under realistic workload dynamics, we select representative bottleneck targets based on structural ranking and treat each selected target as the monitored function whose resource time series drives forecasting and scaling decisions. This design isolates the structural prioritization mechanism from workload generation, allowing the dependency-aware control logic to be evaluated using real performance traces without requiring proprietary serverless call-graph data. Table 3 presents representative sample records from the dependency analysis dataset, illustrating the file and class identifiers along with their associated dependency metrics.

5.3 Bottleneck identification and threshold-based decisions

The dependency graph constructed from the structural dataset is analyzed to identify high-impact nodes using weighted degree centrality. Figure 4 presents the top bottleneck candidates ranked by normalized centrality score. The most

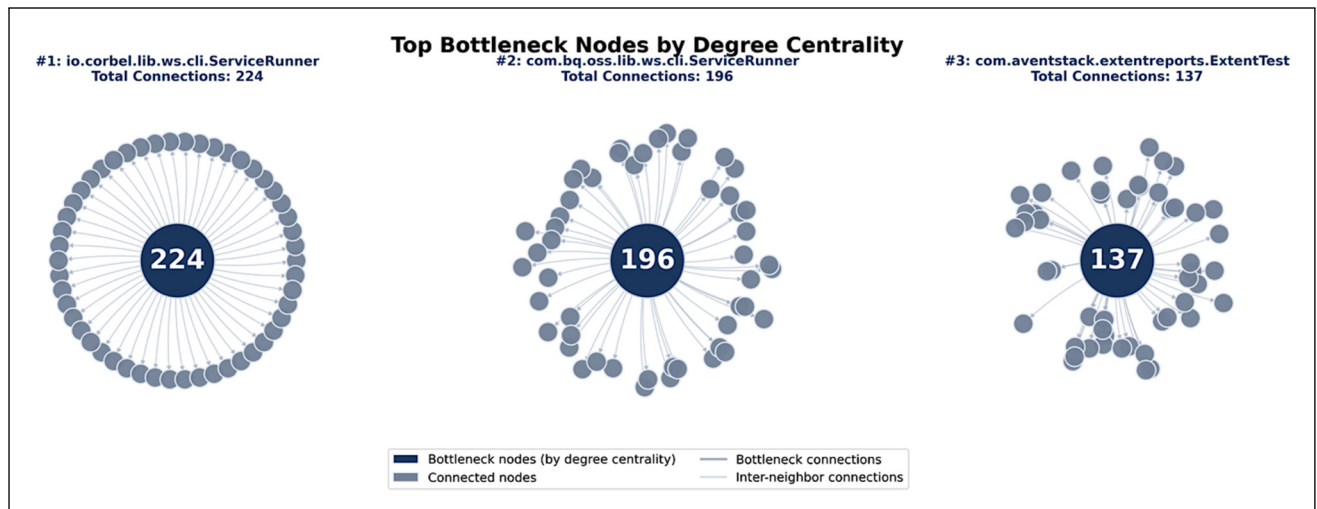


Figure 4. Bottleneck candidates ranked by normalized degree centrality.

connected node exhibits 224 total dependency connections, followed by nodes with 196 and 137 connections. These nodes interact with a large number of components and therefore are structurally positioned to influence multiple execution paths. In the evaluation pipeline, the highest-ranked node is selected as the bottleneck watch target. Its associated resource time series is used for subsequent forecasting and scaling experiments. To generate baseline scaling decisions, runtime metrics of the selected bottleneck are compared against percentile-based thresholds derived from historical behavior. Upper thresholds correspond to the 95th percentile for CPU utilization, memory usage, and execution time, while available memory uses the 5th percentile to detect critically low capacity. Lower thresholds are defined at the 70th percentile for CPU and memory utilization and the 10th percentile for available memory, introducing hysteresis to reduce oscillatory behavior. Scaling actions are produced using the three-way logic defined in Section 4.3: scale-up is triggered if any metric exceeds its upper bound, scale-down occurs only when all metrics fall below their lower bounds, and hold is issued otherwise. Grounding thresholds in empirical distributions reduces sensitivity to transient fluctuations while preserving responsiveness to sustained load increases. In addition to serving as a baseline controller, this threshold mechanism provides consistent action labels for supervised model training and for mapping unsupervised clusters to discrete scaling decisions.

5.4 Unsupervised learning evaluations

We evaluate K-Means, SOM, and PCA-based clustering to assess whether unlabeled grouping of resource traces can recover scaling decisions without supervision. Cluster assignments are post-processed using majority voting to map each cluster to one of the three scaling actions (scale-down, hold, scale-up), ensuring comparability with supervised classifiers. Figure 5 presents K-Means clustering visualized using t-SNE. While the projection reveals partially distinguishable groups, the clusters do not align consistently with scaling labels, resulting in an accuracy of 49.5%. Figure 6 shows SOM-based clustering. The resulting clusters exhibit substantial overlap between scaling categories, and decision alignment accuracy drops to 21.6%. Figure 7 presents K-Means clustering applied after PCA dimensionality reduction. The projection fails to meaningfully separate scaling behaviors, producing near-random performance with an accuracy of 0.6%. These results demonstrate that unsupervised pattern discovery, even when visually separable in reduced-dimensional space, does not reliably recover the control logic required for autoscaling. Structural grouping alone is insufficient to capture the directional decision boundaries needed for scale-up and scale-down actions.

Table 4 compares unsupervised clustering approaches with supervised neural models (MLP, LSTM, and CNN) under identical input features. The supervised models achieve consistently high predictive performance. The LSTM obtains the highest average accuracy (99.06%), followed by MLP (98.75%) and CNN (98.50%). Precision, recall, and F1-scores exceed 0.98 across models, indicating stable classification across scaling categories. Error metrics (MAE and MSE) are correspondingly low.

In contrast, the strongest unsupervised baseline (K-Means with t-SNE visualization) achieves only 49.5% accuracy, while SOM and PCA-based clustering perform substantially worse. The large performance gap of nearly 50% points indicates that labeled supervision is critical for accurate autoscaling decision generation. These findings confirm that

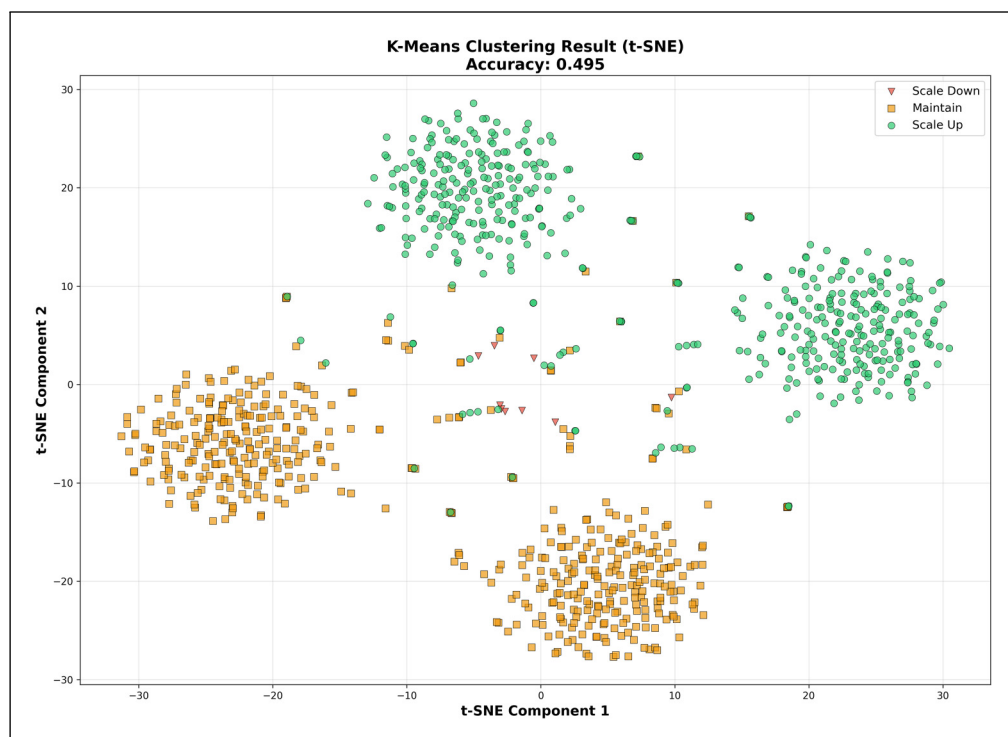


Figure 5. K-Means clustering results visualized with t-SNE.

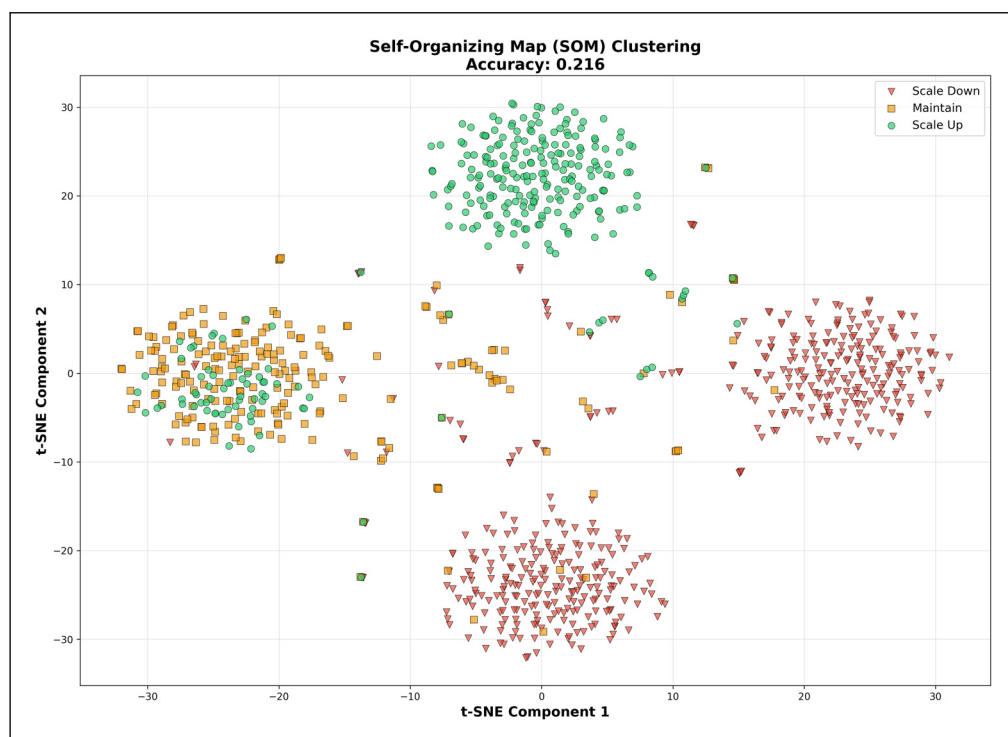


Figure 6. SOM clustering visualized with t-SNE.

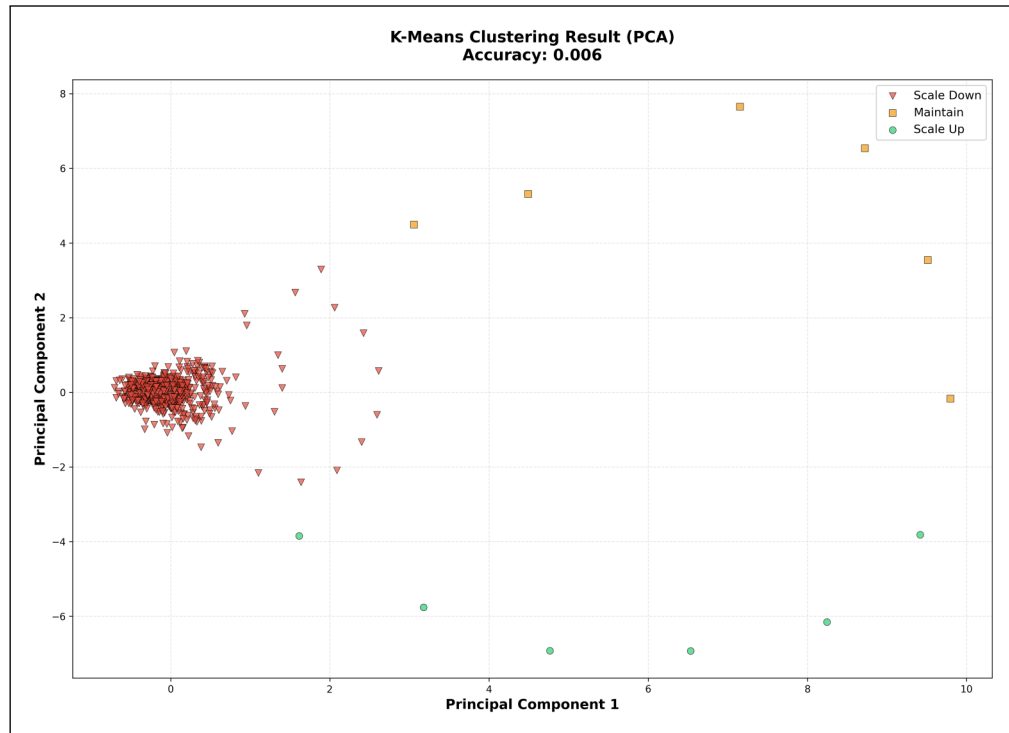


Figure 7. K-Means clustering results visualized with PCA.

Table 4. Comparative performance analysis of supervised and unsupervised approaches for autoscaling decision prediction.

Approach	Model	MAE	MSE	Precision	Recall	F1-Score	Accuracy (%)
Supervised	MLP	0.02	0.02	0.98	0.99	0.98	98.75
	LSTM	0.01	0.01	0.98	0.99	0.99	99.06
	CNN	0.02	0.04	0.98	0.99	0.98	98.50
Unsupervised	K-Means (t-SNE)	—	—	—	—	—	49.5
	SOM	—	—	—	—	—	21.6
	K-Means (PCA)	—	—	—	—	—	0.6

clustering-based pattern discovery cannot substitute for predictive modeling in time-sensitive scaling control. Consequently, the final controller relies on supervised forecasting combined with consensus aggregation rather than unsupervised grouping.

5.5 Supervised learning evaluations

In the supervised phase, autoscaling is formulated as a three-class classification problem using labels generated by the adaptive threshold mechanism described in Section 4.3. The dataset is divided into 80% training, 10% validation, and 10% for testing. To assess generalization, five-fold cross-validation is performed. Model performance is evaluated using accuracy, precision, recall, F1-score, MAE, MSE, and RMSE. Figure 8 represents the training and validation curves for the MLP, LSTM, and CNN models over 100 epochs. All three architectures converge rapidly during early training and stabilize thereafter. Training and validation curves remain closely aligned across accuracy, MAE, and MSE, indicating minimal overfitting. Among the models, the LSTM exhibits the most stable convergence with consistently smooth validation curves. The MLP shows slightly greater variability, while the CNN maintains stable behavior but with marginally higher final error values. These learning dynamics indicate that all models are well-regularized and suitable for real-time inference.

Figure 9 compares final error metrics across models. The LSTM achieves the lowest MAE, MSE, and RMSE values, confirming its strong temporal modeling capability. The MLP demonstrates competitive performance with slightly higher error values. The CNN performs well overall but exhibits higher variability in certain folds. The five-fold cross-validation results in Tables 5–7 further support these findings. The LSTM consistently achieves the lowest error and highest stability

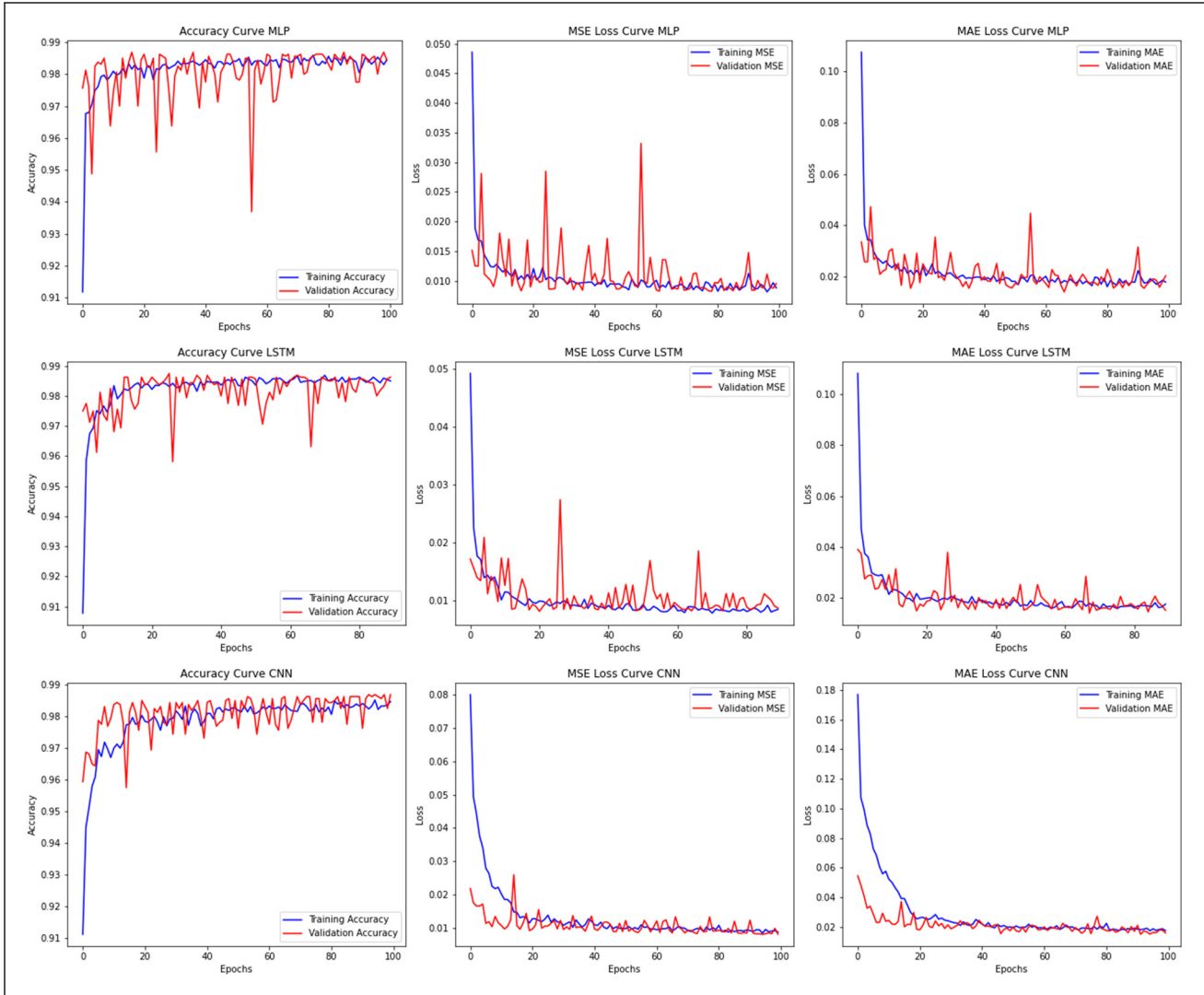


Figure 8. Training and validation performance of the supervised learning models (MLP – first row, LSTM – second row, CNN – third row) over 100 epochs. The columns report classification accuracy, MSE, and MAE. All models demonstrate stable convergence with closely aligned training and validation curves, with the LSTM exhibiting the most consistent and robust performance.

across folds. Although all supervised models outperform unsupervised baselines by a large margin, the LSTM provides the strongest standalone predictor.

Tables 8 and 9 compare individual model predictions with the performance-weighted probabilistic ensemble for representative samples at time t and one-step-ahead forecasts $t + 1$. While individual predictors occasionally deviate from actual values, the ensemble consistently reduces extreme deviations by weighting model outputs according to validation performance. When a single model underestimates or overshoots resource demand, the aggregated prediction moderates the error through probabilistic averaging. This behavior demonstrates that consensus aggregation reduces reliance on any single model and improves stability under workload variability. The ensemble prediction tracks actual resource values more consistently than individual models, supporting the design choice of performance-weighted fusion in the final controller.

5.6 Comparison with related forecasting methods

To position the proposed framework within the current literature, we compare it against recent neural forecasting architectures, including hybrid and ensemble approaches (Javeed et al., 2025; Ouhamme et al., 2021; Sabyasachi et al., 2024; Taha et al., 2024; Vaghassia et al., 2025). All methods are evaluated using the same workload traces and performance metrics to ensure a fair comparison. Unlike hybrid architectures that merge multiple networks into a single composite

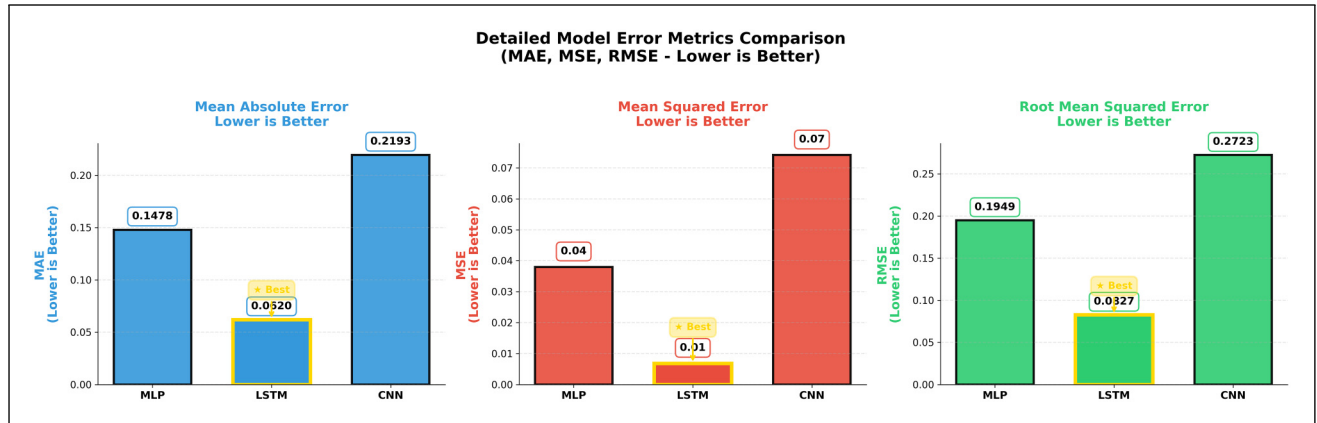


Figure 9. Comparative evaluation of prediction error metrics for the MLP, LSTM, and CNN supervised models. The models are assessed using MAE, MSE, and RMSE, where lower values indicate better performance.

Table 5. Five-fold cross-validation performance of the MLP model, including MAE, MSE, FI-score, recall, precision, and accuracy across folds and averaged results.

	Performance analysis of MLP				Error analysis of MLP	
	Accuracy	Precision	Recall	FI-Score	MSE	MAE
Fold 1	98.40%	0.98	0.98	0.98	0.04	0.02
Fold 2	99.00%	0.98	0.99	0.99	0.02	0.01
Fold 3	98.75%	0.98	0.99	0.98	0.03	0.02
Fold 4	98.90%	0.98	0.99	0.99	0.02	0.01
Fold 5	98.70%	0.98	0.99	0.98	0.02	0.02
Average	98.75%	0.98	0.99	0.98	0.02	0.02

Table 6. Five-fold cross-validation performance of the LSTM model, including MAE, MSE, FI-score, recall, precision, and accuracy across folds and averaged results.

	Performance analysis of LSTM				Error analysis of LSTM	
	Accuracy	Precision	Recall	FI-Score	MSE	MAE
Fold 1	99.20%	0.99	0.99	0.99	0.01	0.01
Fold 2	98.95%	0.98	0.99	0.99	0.02	0.01
Fold 3	99.05%	0.98	0.99	0.99	0.01	0.01
Fold 4	99.05%	0.98	0.99	0.99	0.01	0.01
Fold 5	99.05%	0.98	0.99	0.99	0.01	0.01
Average	99.06%	0.98	0.99	0.99	0.01	0.01

Table 7. Five-fold cross-validation performance of the CNN model, including MAE, MSE, FI-score, recall, precision, and accuracy across folds and averaged results.

	Performance analysis of CNN				Error analysis of CNN	
	Accuracy	Precision	Recall	FI-Score	MSE	MAE
Fold 1	98.75%	0.98	0.99	0.98	0.02	0.02
Fold 2	98.65%	0.98	0.99	0.98	0.03	0.02
Fold 3	97.15%	0.97	0.97	0.97	0.09	0.05
Fold 4	99.00%	0.98	0.99	0.99	0.02	0.01
Fold 5	98.95%	0.98	0.99	0.99	0.02	0.01
Average	98.5%	0.98	0.99	0.98	0.04	0.02

Table 8. Comparison of actual and predicted resource values at the current time step (t) using MLP, LSTM, CNN, and Bayesian-inspired model averaging.

Bayesian-inspired avg prediction for t	CNN prediction for t	LSTM prediction for t	MLP prediction for t	Actual value for t
9.934422	9.263559	10.364485	10.236974	10.3520393
5.103557	4.7756	5.405163	5.160522	5.2734375
29.304556	29.035881	29.725983	29.177689	29.3650742

Table 9. Comparison of actual and predicted resource values at the next time step ($t + 1$) using MLP, LSTM, CNN, and Bayesian-inspired model averaging.

Bayesian-inspired avg prediction for $t + 1$	CNN prediction for $t + 1$	LSTM prediction for $t + 1$	MLP prediction for $t + 1$	Actual value for $t + 1$
59.63367	46.664474	64.61939	63.02586	64.19699
59.659576	46.92247	64.57348	62.944954	64.19621
59.811844	47.328598	64.62794	63.03141	64.20816

Table 10. Performance comparison of the proposed ensemble approach with representative baseline methods. The proposed method achieves superior predictive accuracy and error reduction while explicitly addressing cold-start effects.

Method	Accuracy	MAE	MSE	RMSE	Cold-Start aware
Proposed Ensemble Approach	0.9988	0.0021	0.0039	0.0403	Yes
MLP-LSTM (Taha et al., 2024)	0.9949	0.0090	0.0168	0.1263	No
ML Models (Javeed et al., 2025)	0.7503	0.4946	0.9843	0.9871	No
ML with Hyperparameter Tuning (Vaghasia et al., 2025)	0.9978	0.0031	0.0049	0.0483	No
CNN-LSTM (Ouhamme et al., 2021)	0.9985	0.0030	0.0060	0.0565	No
Deep CNN-LSTM (Sabyasachi et al., 2024)	0.9962	0.0060	0.0104	0.0906	No

predictor (e.g., CNN-LSTM or MLP-LSTM), the proposed framework maintains model diversity by training independent predictors and aggregating their outputs using a performance-weighted probabilistic ensemble. This design preserves complementary signal representations while reducing sensitivity to noise, local minima, and workload drift. Table 10 summarizes the comparative results. The proposed approach achieves the highest overall accuracy and the lowest MAE, MSE, and RMSE among all evaluated methods. In comparison, the closest competing hybrid model (CNN-LSTM Ouhamme et al., 2021) achieves 99.85% accuracy with a higher MSE (0.0060). Deep CNN-LSTM (Sabyasachi et al., 2024) reports 99.62% accuracy and higher error values across all metrics. The MLP-LSTM hybrid (Taha et al., 2024) achieves 99.49% accuracy but exhibits nearly four times higher MSE than the proposed approach. Traditional machine learning models (Javeed et al., 2025) achieve substantially lower accuracy (75.03%) and significantly higher error rates. The proposed framework improves accuracy by up to 4.85% points over traditional ML baselines and achieves substantial reductions in MSE relative to hybrid deep models. In addition to predictive accuracy, the proposed method explicitly incorporates cold-start awareness within its cost-aware control logic, whereas the compared forecasting models focus solely on prediction performance without integrating scaling control or SLA-aware filtering. These results demonstrate that separating model expertise and reconciling predictions via performance-weighted aggregation yields both improved quantitative accuracy and greater robustness than monolithic hybrid architectures.

5.7 Autoscaler cost impact

We evaluate the economic impact of the proposed controller while maintaining response latency near the target SLA. The controller forecasts near-term demand at one-minute intervals and converts predictions into discrete scaling actions. A cooldown mechanism limits oscillatory behavior, and scale-down decisions are issued only when predicted demand remains low and latency stays within a guard band. Scale-to-zero is applied selectively based on projected economic benefit and anticipated cold-start impact. At each decision step, the total cost is computed using platform-specific pricing models. For function-based services, cost includes request charges and memory-time billing, whereas container-based services use vCPU- and memory-second pricing. The cost formulation incorporates overload penalties and SLA violation

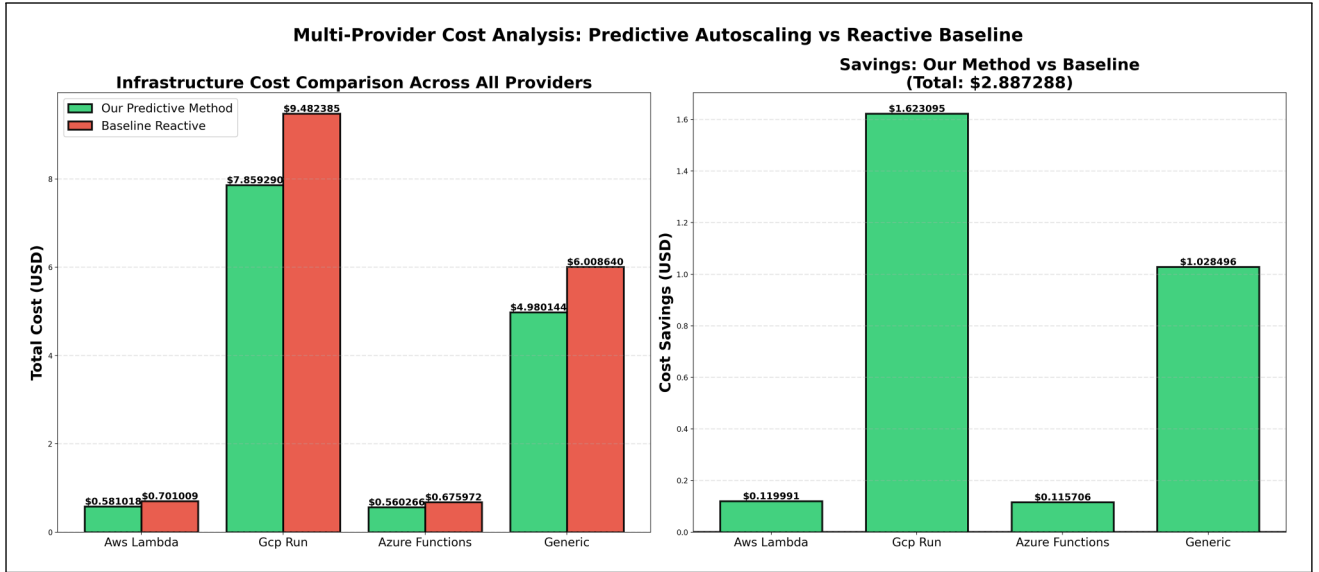


Figure 10. Total infrastructure cost and corresponding savings achieved by the proposed predictive autoscaling method.

indicators. Cold-start effects are modeled indirectly: scale-down actions that increase predicted latency beyond the SLA threshold incur additional penalty, reflecting the expected delay when insufficient warm capacity is available. This penalty discourages aggressive deprovisioning that could otherwise produce economically unfavorable latency spikes.

Figure 10 reports total infrastructure cost and corresponding savings relative to a reactive threshold-based baseline. Across all evaluated platforms, the predictive controller consistently reduces total cost while preserving performance targets. On AWS Lambda, the cost decreases from \$0.70 to \$0.47. On Google Cloud Run, the cost decreases from \$9.48 to \$6.36. On Azure Functions, the cost decreases from \$0.68 to \$0.45. Under a generic container pricing model, the cost decreases from \$6.01 to \$4.03. The aggregated savings across platforms amount to \$5.55 relative to the reactive baseline. These savings result from two complementary effects. First, proactive scale-up mitigates prolonged latency excursions that would otherwise trigger delayed and excessive reactive scaling. Second, controlled scale-down reduces idle replica time while avoiding SLA violations and cold-start penalties. Together, these mechanisms demonstrate that dependency-aware predictive autoscaling can achieve economically efficient resource allocation without compromising responsiveness or stability.

6 Conclusion

This paper presents a dependency-aware autoscaling framework for serverless environments that addresses structural, predictive, and economic limitations of existing approaches. By modeling function interactions as a directed dependency graph and identifying structurally influential bottlenecks using degree centrality, the framework prioritizes scaling actions on components with the greatest end-to-end impact. Near-term resource demand is predicted using lightweight supervised models (MLP, LSTM, and CNN), and their outputs are reconciled through a performance-weighted probabilistic ensemble to improve stability and reduce sensitivity to model-specific bias. Our approach incorporates cold-start awareness and cost-comparison logic to balance latency guarantees with operational efficiency. Experimental results demonstrate that supervised learning substantially outperforms unsupervised clustering for autoscaling decision generation. The proposed ensemble approach achieves higher predictive accuracy and lower error than representative hybrid forecasting models while maintaining stable training dynamics. End-to-end experiments across multiple pricing models show consistent cost reductions, without compromising SLA targets. Overall, this work shows that integrating dependency-aware bottleneck targeting, supervised multi-model forecasting, and cost-aware control yields a robust and economically efficient solution for practical serverless autoscaling.

7 Future work

Although the proposed framework advances dependency-aware and ensemble-based autoscaling, several directions remain for further improvement in adaptability, scalability, and interpretability. These extensions aim to evolve the framework

from predictive autoscaling toward more adaptive and intelligent resource management in next-generation serverless systems.

1. Federated learning may enable collaborative model training across multiple serverless platforms without sharing raw workload data. This approach would allow platforms to benefit from shared knowledge while preserving privacy. Open challenges include handling workload heterogeneity, personalizing global models to local behavior, and ensuring secure and stable aggregation of distributed updates.
2. Deploying the proposed framework on real FaaS platforms such as AWS Lambda or Knative would enable evaluation under fully operational conditions. Integrating dependency inference, workload forecasting, and actuation within live systems would support analysis of long-term stability, convergence behavior, and tail latency performance under bursty and non-stationary demand.
3. Quantum-inspired optimization offers a potential avenue for improving scalability in large dependency graphs. By formulating scaling decisions as multi-objective optimization problems, such heuristics may provide efficient approximations for jointly optimizing latency, cost, and resource utilization in complex systems.
4. Graph neural networks (GNNs) could extend the framework to dynamic dependency modeling. Unlike static graph metrics, temporal GNNs may learn evolving interaction patterns and detect emerging critical paths, enabling earlier bottleneck identification and more proactive resource allocation.
5. Incorporating causal inference into the autoscaling logic may enhance both robustness and interpretability. By identifying which factors directly cause latency or overload, rather than relying only on correlated signals, the controller can make more reliable scaling decisions when workload patterns change and provide clearer explanations for its actions.

ORCID iD

Mehrdad Ashtiani  <https://orcid.org/0000-0002-4574-1545>

Ethical approval and consent to participate

This article does not contain any studies with human participants or animals performed by any of the authors.

Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

Declaration of conflicting interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Availability of data and material

Derived data supporting the findings of this study are publicly available in a GitHub repository and can be shared upon request.

References

- Abdelsamea MM, Gnecco G and Gaber MM (2014) A survey of SOM-based active contour models for image segmentation. In: *Advances in Intelligent Systems and Computing*, vol. 295. Springer Verlag, pp. 293–302. https://doi.org/10.1007/978-3-319-07695-9_28.
- Abdi H and Williams LJ (2010) Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics* 2: 433–459.
- Ahmed M, Seraj R and Islam SMS (2020) The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics (Switzerland)* 9: 1–12.
- Al-Dulaimy A, Taheri J, Kassler A, et al. (2022) Multiscaler: A multi-loop auto-scaling approach for cloud-based applications. *IEEE Transactions on Cloud Computing* 10: 2769–2786.
- Almeida V, Arlitt M and Rolia J (2002) Analyzing a web-based system's performance measures at multiple time scales. *ACM SIGMETRICS Performance Evaluation Review* 30: 3–9.
- Alzubaidi L, Zhang J, Humaidi AJ, et al. (2021) Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data* 8(1): 53.
- Anonymous (2016) Automatic cloud resource scaling algorithm based on long short-term memory recurrent neural network. *International Journal of Advanced Computer Science and Applications* 7. <https://doi.org/10.14569/ijacsa.2016.071236>.
- Anselmi J, Gaujal B and Rebuffi LS (2025) Non-stationary gradient descent for optimal auto-scaling in serverless platforms. *IEEE Transactions on Networking* 33: 1574–1587.
- Arbel J, Pitas K, Vladimirova M, et al. (2023) A primer on bayesian neural networks: review and debates.

- Aslanpour MS, Toosi A, Cheema M, et al. (2024) Load balancing for heterogeneous serverless edge computing: A performance-driven and empirical approach. *Future Generation Computer Systems* 154: 266–280.
- Bazrafshan O, Ehteram M, Dashti Latif S, et al. (2022) Predicting crop yields using a new robust Bayesian averaging model based on multiple hybrid ANFIS and MLP models: Predicting crop yields using a new robust Bayesian averaging model. *Ain Shams Engineering Journal* 13(5): 101724.
- Berahmand K, Saberi-Movahed F, Sheikhpour R, et al. (2025) A comprehensive survey on spectral clustering with graph structure learning.
- Bibal Benifa JV and Dejeu D (2019) RLPAS: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications* 24: 1348–1363.
- Chang YI (2025) A survey: Potential dimensionality reduction methods.
- Chen T, Bahsoon R and Yao X (2019) A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys* 51. <https://doi.org/10.1145/3190507>.
- Chitsaz B, Khonsari A, Moradian M, et al. (2023) Scaling power management in cloud data centers: a multi-level continuous-time MDP approach. <https://doi.org/10.48550/arXiv.2108.01292>.
- Dashtbani M and Tahvildari L (2025) Key considerations for auto-scaling: lessons from benchmark microservices.
- Denton PB, Parke SJ, Tao T, et al. (2021) Eigenvectors from eigenvalues: A survey of a basic identity in linear algebra. <https://doi.org/10.1090/bull/1722>.
- Ding Y, Li C, Cai Z, et al. (2025) A dynamic interval auto-scaling optimization method based on informer time series prediction. *IEEE Access* 13: 14572–14583.
- Dutreilh X, Rivierre N, Moreau A, et al. (2010) From data center resource allocation to control theory and back. In: 2010 IEEE 3rd International Conference on Cloud Computing. <https://doi.org/10.1109/CLOUD.2010.55>.
- Elngar AA, Arafa M, Fathy A, et al. (2021) Image classification based on CNN: A survey. *Journal of Cybersecurity and Information Management (JCIM)* 6: 18.
- Fang W, Lu ZH, Wu J, et al. (2012) RPPS: A novel resource prediction and provisioning scheme in cloud data center. In: 2012 IEEE 9th International Conference on Services Computing (SCC), pp. 609–616. <https://doi.org/10.1109/SCC.2012.47>.
- Florida S, Liu Y and Weisberg RH (n.d) A review of self-organizing map applications in meteorology and a review of self-organizing map applications in meteorology and oceanography oceanography scholar commons citation scholar commons citation “a review of self-organizing map applications in meteorology 14 a review of self-organizing map applications in meteorology and oceanography”.
- Fog JW, Moller JJT, Jensen TM, et al. (2025) Comparing neural and statistical time-series models for proactive auto-scaling in kubernetes. In: Proceedings - 19th IEEE International Conference on Service-Oriented System Engineering, SOSE 2025, pp. 151–161. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/SOSE67019.2025.00022>.
- Fragoso TM, Bertoli W and Louzada F (2018) Bayesian model averaging: A systematic review and conceptual classification. *International Statistical Review* 86: 1–28.
- Gambi A and Toffetti G (2012) Modeling cloud performance with kriging. In: 34th International Conference on Software Engineering (ICSE). <https://doi.org/10.1109/ICSE.2012.6227075>.
- Ginarsa NAN and Santoso BJ (2025) Intelligent kubernetes autoscaling through generative AI-driven workload predictions. In: Proceedings - 2025 4th International Conference on Electronics Representation and Algorithm: Artificial Intelligence: Creating Tomorrow's World Today, ICERA 2025, pp. 400–404. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/ICERA66156.2025.11087276>.
- Golshani E and Ashtiani M (2021) Proactive auto-scaling for cloud environments using temporal convolutional neural networks. *Journal of Parallel and Distributed Computing* 154: 119–141.
- Govindarajan K and Tienne AD (2023) Resource management in serverless computing: review, research challenges, and prospects. In: 12th International Conference on Advanced Computing (ICoAC). IEEE. <https://doi.org/10.1109/ICoAC59537.2023.10249574>.
- Han R, Guo L, Ghanem MM, et al. (2012) Lightweight resource scaling for cloud applications. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). <https://doi.org/10.1109/CCGrid.2012.52>.
- Hinne M, Gronau QF, van den Bergh D, et al. (2020) A conceptual Introduction to Bayesian model averaging. *Advances in Methods and Practices in Psychological Science* 3: 200–215.
- Hu W, Hu G, Balakrishnan M, et al. (2025) Marlin: Efficient coordination for autoscaling cloud DBMS. *Proceedings of the ACM on Management of Data* 3: 1–28.
- Jafarnejad Ghomi E, Rahmani AM and Qader NN (2019) Applying queue theory for modeling of cloud computing: A systematic review. *Concurrency and Computation* 31(17): e5186.
- Javeed A, Borg A, Grahm H, et al. (2025) Improving cloud efficiency: A machine learning-based stacking model for CPU utilization prediction. In: Proceedings - 2025 8th International Conference on Data Science and Machine Learning Applications, CDMA 2025, pp. 120–125. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/CDMA61895.2025.00026>.
- Jonas E, Schleier-Smith J, Sreekanti V, et al. (2019) Cloud programming simplified: a berkeley view on serverless computing.

- Khan ME (2023) The Bayesian Learning Rule. vol. 24.
- Khatua S, Ghosh A and Mukherjee N (2010) Optimizing the utilization of virtual resources in cloud environment. In: 2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems (VECIMS), pp. 82–87. <https://doi.org/10.1109/VECIMS.2010.5609349>.
- Koperek P and Funika W (2012) Dynamic business metrics-driven resource provisioning in cloud environments. In: Lecture Notes in Computer Science, vol. 7204, pp. 171–180, Springer. https://doi.org/10.1007/978-3-642-31500-8_18.
- Kumar J, Goomer R and Singh AK (2018) Long short term memory recurrent neural network (LSTM-RNN) based workload forecasting model for cloud datacenters. *Procedia Computer Science* 125: 676–682.
- Landi F, Baraldi L, Cornia M, et al. (2021) Working memory connections for LSTM. <https://doi.org/10.1016/j.neunet.2021.08.030>.
- Li B, Xia T, Wang W, et al. (2025) FluidEdge: Expediting serverless machine learning inference via bottleneck-aware auto-scaling on Edge SoCs. *IEEE Transactions on Mobile Computing* 24(12): 13586–13599.
- Lim HC, Babu S and Chase JS (2010) Automated control for elastic storage. In: 7th USENIX Conference on Networked Systems Design and Implementation (NSDI).
- Ling LS and Weiling CT (2025) Enhancing segmentation: A comparative study of clustering methods. *IEEE Access* 13: 47418–47439.
- Lorido-Botran T, Miguel-Alonso J and Lozano JA (2013) Comparison of auto-scaling techniques for cloud environments.
- Lu Y and Salem FM (2017) Simplified gating in long short-term memory (LSTM) recurrent neural networks. CoRR. abs/1701.03441.
- Ma~kiewicz A and Ratajczak W (1993) Principal components analysis (PCA)*. vol. 19.
- Madsen KV (2022) Performance Data. <https://doi.org/10.5281/zenodo.14262334>.
- Mampage A, Karunasekera S and Buyya R (2021) Deadline-aware dynamic resource management in serverless computing environments. In: 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 483–492. IEEE. <https://doi.org/10.1109/CCGrid51090.2021.00058>.
- Mampage A, Karunasekera S and Buyya R (2022) A holistic view on resource management in serverless computing environments: taxonomy and future directions. *ACM Computing Surveys* 54(11s): 1–3.
- Mampage A, Karunasekera S and Buyya R (2023) A deep reinforcement learning based algorithm for time and cost optimized scaling of serverless applications.
- Manner J (2023) A structured literature review approach to define serverless computing and function as a service. In: IEEE International Conference on Cloud Computing, CLOUD, vol. 2023- July, pp. 516–522. IEEE Computer Society. <https://doi.org/10.1109/CLOUD60044.2023.00068>.
- Maurer M, Brandic I, Emeakaroha V, et al. (2011) Enacting SLAs in clouds using rules. In: Euro-Par 2011: Parallel Processing, pp.147–152.
- Mittal M, Praveen Gujjar J, Guru Prasad MS, et al. (2024) Dimensionality reduction using UMAP and TSNE technique. In: 2nd IEEE International Conference on Advances in Information Technology, ICAIT 2024 – Proceedings. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/ICAIT61638.2024.10690797>.
- Nadjaran Toosi A, Son J, Chi Q, et al. (2019) ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds. *Journal of Systems and Software* 152: 108–119.
- Nunes JPKS, Nejati S, Sabetzadeh M, et al. (2024) Self-adaptive, requirements-driven autoscaling of microservices. In: Proceedings - 2024 IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2024, pp. 168–174. Association for Computing Machinery, Inc. <https://doi.org/10.1145/3643915.3644094>.
- Ouhame S, Hadi Y and Ullah A (2021) An efficient forecasting approach for resource utilization in cloud data center using CNN-LSTM model. *Neural Computing & Applications* 33: 10043–10055.
- Pandey V (2025) Cost-Optimized predictive autoscaling of cloud resources using game-theoretic queuing theory. In: Conference Proceedings - IEEE SOUTHEASTCON, pp. 481–487. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/SoutheastCon56624.2025.10971700>.
- Popescu M-C and Balas VE (n.d) Multilayer Perceptron and Neural Networks. <https://doi.org/10.5555/1639537.1639542>.
- Pulver A and Lyu S (2017) LSTM with working memory. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 845–851. IEEE. <https://doi.org/10.1109/IJCNN.2017.7965940>.
- Purwono A, Ma'arif , Rahmaniar W, et al. (2022) Understanding of convolutional neural network (CNN): A review. *International Journal of Robotics and Control Systems* 2: 739–748.
- Qu X, Yang L, Guo K, et al. (2021) A survey on the development of self-organizing maps for unsupervised intrusion detection. *Mobile Networks and Applications* 26: 808–829.
- Rana A, Singh Rawat A, Bijalwan A, et al. (2018) Application of multi layer (perceptron) artificial neural network in the diagnosis system: A systematic review. In: 2018 International Conference on Research in Intelligent and Computing in Engineering (RICE), pp. 1–6. IEEE. <https://doi.org/10.1109/RICE.2018.8509069>.
- Rao J, Bu X, Xu C-Z, et al. (2009) VCONF: A reinforcement learning approach to virtual machines auto-configuration. In: 6th International Conference on Autonomic Computing (ICAC), pp. 137–146. ACM. <https://doi.org/10.1145/1555228.1555263>.

- Robino L, Garí Y, Pacini E, et al. (2025) Reinforcement learning-based cloud autoscaler initialization via evolutionary algorithms. <https://doi.org/10.1145/3712255>.
- Rokach L and Maimon O (2006) Clustering methods. In: *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag, pp. 321–352. https://doi.org/10.1007/0-387-25465-x_15.
- Roy N, Dubey A and Gokhale A (n.d) Efficient autoscaling in the cloud using predictive models for workload forecasting.
- Sabyasachi AS, Sahoo BM and Ranganath A (2024) Deep CNN and LSTM approaches for efficient workload prediction in cloud environment. *Procedia Computer Science* 235: 2651–2661.
- Sahu M and Dash R (2021) A survey on deep learning: Convolution neural network (cnn). In: *Smart Innovation, Systems and Technologies*, vol. 153. Springer Science and Business Media Deutschland GmbH, pp. 317–325. https://doi.org/10.1007/978-981-15-6202-0_32.
- Salah K, Elbadawi K and Boutaba R (2016) An analytical model for estimating cloud resources of elastic services. *Journal of Network and Systems Management* 24: 285–308.
- Santos J, Reppas E, Wauters T, et al. (2025) Can reinforcement learning be generalized for efficient auto-scaling in containerized clouds? In: *Proceedings of IEEE/IFIP Network Operations and Management Symposium 2025, NOMS 2025*. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/NOMS57970.2025.11073717>.
- Schuler L, Jamil S and Kühl N (2020) AI-based resource allocation: reinforcement learning for adaptive auto-scaling in serverless environments.
- Semerikov S, Zubov D, Kupin A, et al. (2024) Models and technologies for autoscaling based on machine learning for microservices architecture.
- Sfakianakis Y, Marazakis M, Kozanitis C, et al. (2022) Latest: Vertical elasticity for millisecond serverless execution. In: *22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 879–885. <https://doi.org/10.1109/CCGrid54584.2022.00105>.
- Shafiei H, Khonsari A and Mousavi P (2022) Serverless computing: A survey of opportunities, challenges, and applications. *ACM Computing Surveys* 54: 1–32.
- Siami-Namini S, Tavakoli N and Namin AS (2019) The performance of LSTM and BiLSTM in forecasting time series. In: *2019 IEEE International Conference on Big Data (Big Data)*, pp. 3285–3292. IEEE. <https://doi.org/10.1109/BigData47090.2019.9005997>.
- Singh J and Banerjee R (2019) A study on single and multi-layer perceptron neural network. In: *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 35–40. IEEE. <https://doi.org/10.1109/ICCMC.2019.8819775>.
- Smagulova K and James AP (2019) A survey on LSTM memristive neural network architectures and applications. *European Physical Journal: Special Topics* 228: 2313–2324.
- Song S, Tong H, Meng C, et al. (2024) Funcscaler: Cold-start-aware holistic autoscaling for serverless resource management. In: *Proceedings of the IEEE International Conference on Web Services, ICWS*, pp. 1036–1047. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/ICWS62655.2024.00122>.
- Taha MB, Sanjalawe Y, Al-Daraiseh A, et al. (2024) Proactive auto-scaling for service function chains in cloud computing based on deep learning. *IEEE Access* 12: 38575–38593.
- Tari M, Ghobaei-Arani M and Pouramini J (2024) Auto-scaling mechanisms in serverless computing: A comprehensive review. *Computer Science Review* 53: 100650.
- Tournaire T, Castel-Taleb H and Hyon E (2023) Efficient computation of optimal thresholds in cloud auto-scaling systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 8(4): 1–3.
- Vaghasia P, Goswami A, Patel D, et al. (2025) Improving predictive accuracy with cloud-based machine learning models for big data analytics. In: *2025 International Conference on Computing Technologies, ICOCT 2025*. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/ICOCT64433.2025.11118785>.
- Valkenborg D, Geubbelmans M, Rousseau AJ, et al. (2023) Supervised learning. *American Journal of Orthodontics and Dentofacial Orthopedics* 164: 146–149.
- Villela D, Pradhan P and Rubenstein D (2004) Provisioning servers in the application tier for E-commerce systems. In: *12th International Workshop on Quality of Service (IWQoS)*. <https://doi.org/10.1109/iwqos.2004.1309357>.
- Wang X, Jin Y, Schmitt S, et al. (2023) Recent advances in Bayesian optimization. *ACM Computing Surveys* 55(13s): 1–36.
- Wang Z, Zhu S, Li J, et al. (2024) Deepscaling: Autoscaling microservices with stable CPU utilization for large scale production cloud systems. *IEEE/ACM Transactions on Networking* 32: 3961–3976.
- Wen J, Chen Z, Jin X, et al. (2023) Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* 32: 1–61.
- Wen J, Chen Z and Liu X (2022) A literature review on serverless computing.
- Xu L, Saxena D, Yadwadkar NJ, et al. (2023) Dirigo: Self-scaling stateful actors for serverless real-time data processing.
- Yao K, Cohn T, Vylomova K, et al. (n.d) Depth-Gated LSTM.

- Zhang Z, Wang T, Li A, et al. (2022) Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning. In: Proceedings - 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC 2022, pp. 866–871. Institute of Electrical and Electronics Engineers Inc. <https://doi.org/10.1109/COMPSAC54236.2022.00137>.
- Zhang Y, Zhang F, Yang Z, et al. (2023) What and how does in-context learning learn? Bayesian model averaging, parameterization, and generalization.
- Zhao J, Huang F, Lv J, et al. (2020) Do RNN and LSTM have long memory?.
- Zhao L, Yang Y, Li Y, et al. (2021) Understanding, predicting and scheduling serverless workloads under partial interference. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery. <https://doi.org/10.1145/3458817.3476215>.