ILP Optimized LSTM-based Autoscaling and Scheduling of Containers in Edge-cloud Environment

Shivan Singh, Narayan D.G., Sadaf Mujawar, G.S. Hanchinamani, and P.S. Hiremath

KLE Technological University, Hubballi, Karnataka, India

https://doi.org/10.26636/jtit.2025.2.2088

Abstract — Edge computing is a decentralized computing paradigm that brings computation and data storage closer to data sources, enabling faster processing and reduced latency. This approach is critical for real-time applications, but it introduces significant challenges in managing resources efficiently in edge-cloud environments. Issues such as increased response times, inefficient autoscaling, and suboptimal task scheduling arise due to the dynamic and resource-constrained nature of edge nodes. Kubernetes, a widely used container orchestration platform, provides basic autoscaling and scheduling mechanisms, but its default configurations often fail to meet the stringent performance requirements of edge environments, especially in lightweight implementations like KubeEdge. This work presents an ILP-optimized, LSTM-based approach for autoscaling and scheduling in edge–cloud environments. The LSTM model forecasts resource demands using both real-time and historical data, enabling proactive resource allocation, while the integer linear programming (ILP) framework optimally assigns workloads and scales containers to meet predicted demands. By jointly addressing auto-scaling and scheduling challenges, the proposed method improves response time and resource utilization. The experimental setup is built on a KubeEdge testbed deployed across 11 nodes (1 cloud node and 10 edge nodes). Experimental results show that the ILP-enhanced framework achieves a 12.34% reduction in response time and a 7.85% increase in throughput compared to the LSTM-only approach.

Keywords — autoscaling, edge computing, ILP optimization, Kubernetes, LSTM, resource efficiency, scheduling, throughput

1. Introduction

Edge computing represents an approach to data processing that enables computation and storage closer to the data source than using centralized cloud servers. This decentralized model improves real-time data analysis, reduces latency, and improves resource utilization, making it suitable for applications like autonomous systems, smart cities, and industrial automation.

The unique requirements of edge computing environments, including low latency responses and efficient resource utilization, create significant challenges in workload management and resource optimization.

KubeEdge is an open-source framework designed to extend Kubernetes functionality to edge computing environments, enabling efficient management of containerized applications across distributed edge nodes. Bridges the gap between cloud infrastructure and edge devices, facilitating seamless deployment and orchestration of workloads in resource-constrained and geographically dispersed locations.

The architecture of KubeEdge includes components optimized for edge scenarios, such as the CloudCore module, which manages edge node control, configuration, and communication with the Kubernetes API server at the cloud level, and the EdgeCore module, which handles application deployment, resource monitoring, and local decision-making at the edge, reducing dependency on continuous cloud connectivity.

Additionally, KubeEdge incorporates an edge message bus for real-time communication between devices and applications, requiring low latency and high responsiveness. By enhancing Kubernetes with edge-specific capabilities, KubeEdge provides a robust platform for deploying scalable and reliable applications in distributed environments.

Autoscaling completes scheduling by dynamically adjusting the number of container replicas to match workload demands. The Kubernetes HorizontalPod Autoscaler (HPA) primarily relies on metrics such as CPU and memory usage to scale resources. For edge computing, network traffic information can play an important role in reducing response time [1]. Although effective in static or predictable workloads, this approach struggles in dynamic edge environments characterized by unpredictable workload patterns.

Proactive scaling mechanisms, using predictive models such as long-short-term memory (LSTM) networks, offer a promising solution [2]. By analyzing historical and real-time metrics, LSTM models can predict future resource demands, enabling preemptive scaling decisions. Reduce resource underutilization and overprovisioning and also ensure timely responses to workload fluctuations.

Scheduling in edge computing plays an important role in efficiently assigning tasks to nodes while minimizing latency and balancing workloads across distributed resources. Unlike traditional cloud environments, where computational resources are large, edge nodes operate under strict resource limitations. Effective scheduling requires consideration of factors such as network conditions, task relationship, and node heterogeneity. For example, tasks that require real-time processing must

be assigned to nodes closer to the data source to ensure low latency, while less critical tasks can be offloaded to distant nodes or cloud servers [3].

Recent advances have explored machine learning and integer linear programming (ILP) to enhance scheduling efficiency, but their integration with existing Kubernetes architectures remains a challenge. Integrating autoscaling with scheduling in Kubernetes enhances resource management by dynamically adjusting workloads to real-time demands. Autoscaling mechanisms such as HPA scale pod replicas based on resource utilization, while intelligent scheduling ensures optimal workload placement across clusters.

A survey on Kubernetes scheduling algorithms emphasizes the importance of autoscaling-enabled scheduling, advocating for algorithms that adapt to dynamic workloads by integrating autoscaling into the scheduling process [4]. This combined approach improves resource allocation and reduces latency under fluctuating workloads.

This work integrates AI-based autoscaling and scheduling mechanisms customized for Kubernetes-based edge-cloud environments. We propose an ILP-optimized LSTM-based approach that addresses the limitations of existing solutions. Using CPU and memory usage as well as RTT metrics, the LSTM model predicts workloads, enabling proactive scaling decisions. In addition, an ILP-based scheduling algorithm assigns tasks to nodes based on real-time and predicted resource availability, optimizing response time and resource utilization. By integrating these processes, the proposed framework ensures efficient operation in dynamic and resource-constrained environments.

The contributions of this work are as follows:

- an LSTM-based prediction model is proposed to forecast resource utilization such as CPU, memory, and RTT, enabling accurate workload predictions,
- a combined autoscaling and scheduling framework that integrates LSTM-based predictions with ILP-based optimization is proposed,
- the proposed approach is evaluated in a KubeEdge testbed environment to determine improvements in resource utilization, response time, and workload prediction accuracy and then compared to traditional methods.

The rest of the article is as follows. Section 2 reviews related research and background study. Sections 3 and 4 outline the mathematical model, the proposed model, and the algorithms. The results are presented in Section 5, and conclusions are given in Section 6.

2. Background Study

The KubeEdge architecture, as shown in Fig. 1, is designed to seamlessly integrate cloud and edge computing environments. It consists of two main layers: the cloud layer and the edge layer. The cloud layer hosts the master node, which contains critical components like the scheduler, metric server, deployments, and autoscaler. These components ensure that tasks are managed efficiently, that resources are optimally allocated, and that the overall system remains scalable and reliable.

The edge layer, on the other hand, includes multiple worker nodes that are connected through EdgeHub. These worker nodes host applications and devices, providing compute power at the edge of the network closer to end users.

The CloudCore, which operates in the cloud, is a key component of the KubeEdge architecture. It consists of various modules that handle critical communication, synchronization, and management tasks. One of these modules is CloudHub, which acts as the primary gateway for communication between the cloud and the edge nodes. It is responsible for maintaining secure web socket connections and routing messages efficiently. Another important module is the edge controller, which oversees the management of edge nodes and synchronizes pod metadata between the edge and the Kubernetes API server. Additionally, the device controller plays a crucial role in ensuring that device metadata is accurately synchronized between the cloud and edge environments.

The KubeEdge architecture incorporates sophisticated mechanisms to handle varying workloads and dynamic resource requirements. One of the standout features is its autoscaling capability, which ensures that the system can adapt to changes in computational demands. This is particularly important for edge computing scenarios where workloads can fluctuate based on real-time events or user interactions. By dynamically adjusting resources, KubeEdge maintains optimal performance without overprovisioning or underutilizing resources.

Scheduling in KubeEdge is another critical aspect of its architecture. The scheduling framework operates both at the cloud and the edge levels, ensuring that tasks are effectively distributed across available resources. At the cloud level, the master scheduler is responsible for global resource allocation. It evaluates various factors such as resource availability, network conditions, and task priorities to make informed scheduling decisions.

At the edge level, the EdgeCore components handle local scheduling. These components are designed to optimize resource utilization while considering factors such as net-

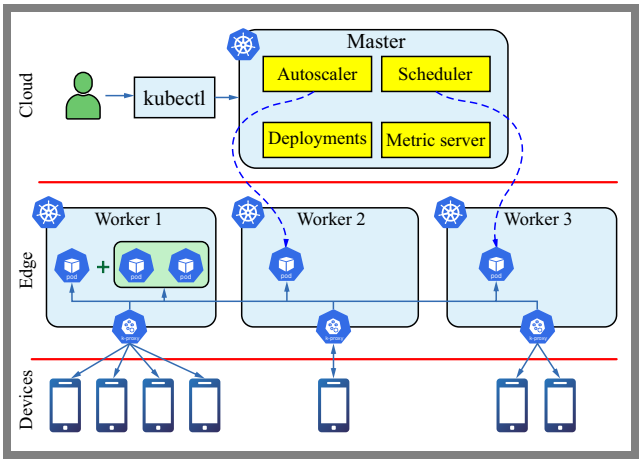

Fig. 1. KubeEdge architecture.

work latency and data locality. The distributed nature of this scheduling framework allows KubeEdge to achieve a balance between centralized control and autonomous operations at the edge.

One of the most important aspects of KubeEdge is its ability to maintain operational consistency even in the face of discontinuous connectivity. Edge environments often operate in challenging conditions where stable network connections cannot be guaranteed. KubeEdge addresses this challenge by ensuring that edge nodes can continue to function autonomously even when disconnected from the cloud. This resilience makes it particularly suitable for scenarios like industrial automation, smart cities, and remote monitoring, where edge devices must continue to operate independently.

2.1. Related Work

In the realm of autoscaling, a hybrid proactive autoscaler optimized for edge computing scenarios has been proposed in [5]. Utilizing a bidirectional long- and short-term memory (Bi-LSTM) based load prediction model, this autoscaler predicts future workloads and performs scaling operations preemptively. Furthermore, an overload compensation algorithm is implemented to ensure quality of service (QoS) degradation due to underprediction, and a hybrid scaling method is applied to simultaneously adjust the number of pods and their resource quota without restarting [5].

In [6], the authors introduce an integrated framework that combines autoscaling and scheduling for edge-cloud environments. This framework dynamically adjusts resources and schedules tasks based on real-time workload analysis, enhancing system performance and resource utilization. Similarly, article [7] presents a proactive autoscaling approach for edge computing systems managed with Kubernetes. By forecasting workloads using multiple user-defined metrics, the proposed proactive pod autoscaler (PPA) scales applications accordingly, outperforming the default autoscaler in both resource utilization efficiency and application performance. These studies highlight the importance of integrating predictive models with resource management strategies to effectively handle dynamic workloads in edge-cloud environments.

Node-aware autoscaling has been extensively explored to address dynamic workloads in edge computing environments. Paper [8] introduces a node-based horizontal pod autoscaler (NHPA) designed for KubeEdge environments. Focuses on the use of resource at the individual node level, allowing dynamic adjustment of pod numbers independently for each node. This ensures optimized pod allocation and seamless scaling, particularly in scenarios where traffic volume fluctuates over time and location, and communication links between edge nodes may be unstable.

Integrating machine learning models with autoscaling has been investigated in [9], while the authors present FedAvg-BiGRU, a proactive autoscaling method in edge computing that combines federated averaging (FedAvg) and multistep prediction using a bidirectional gated recurrent unit (BiGRU). This approach reduces network traffic by exchanging model updates instead of raw data, thereby enhancing resource allocation efficiency. Similarly, [10] proposes a hybrid proactive autoscaler that combines horizontal and vertical scaling based on future workload predictions. This method aims to improve QoS and resource utilization efficiency in Kubernetes clusters.

Predictive workload modeling is crucial to improve autoscaling and scheduling in dynamic environments. In [11], the authors address container job scheduling as a multiobjective optimization problem, proposing a linear programming model to address this issue. They highlight the limitations of traditional approaches in capturing the non-linearities associated with resource usage patterns, suggesting that deep neural networks could offer a more effective solution.

Dynamic resource allocation frameworks have been developed to enhance Kubernetes cluster management. For example, a study [12] proposes a dynamic task offloading framework in KubeEdge-based edge computing environments, utilizing machine learning to optimize resource allocation and ensure data privacy. This approach addresses the challenges of resource limitations and privacy concerns in edge computing scenarios. Similarly, work [13] introduces a proactive hybrid autoscaler designed for edge applications in Kubernetes. By employing a Bi-LSTM based load prediction model, this autoscaler anticipates future workloads, enabling preemptive scaling actions that improve resource utilization and maintain QoS. These frameworks demonstrate the effectiveness of integrating predictive models and machine learning techniques for dynamic resource management in Kubernetes clusters.

Efficient task scheduling is a critical challenge in edge computing environments. In [3], the authors propose a network-based container scheduling approach that considers the various edge node network performance, such as geographical location and network topology, to optimize resource allocation and application performance. Furthermore, [14] investigates the performance of KubeEdge in terms of computational resource distribution and latency between edge nodes. The study reveals that forwarding traffic between edge nodes leads to a degraded throughput and an increased service delay in an edge computing environment. To mitigate this problem, the authors propose a local scheduling scheme that processes user traffic locally at each edge node, enhancing the performance of edge devices.

In edge computing, efficient resource scheduling is crucial to manage the limited computational resources and dynamic workloads characteristic of edge environments. A comprehensive taxonomy of resource scheduling techniques is presented in [15], categorizing approaches based on application scenarios, computational platforms, algorithm paradigms, and optimization objectives. This taxonomy addresses challenges such as heterogeneity, workload dynamics, and the need for real-time processing, providing a structured framework for developing effective scheduling strategies. In addition, a multi-objective optimization algorithm is introduced in [16], with the aim of minimizing latency and maximizing resource utilization in edge systems. This algorithm considers factors such as task allocation, resource availability, and

Tab. 1. Summary of related work and research gaps.

Ref.	Focus area	Methodology	Research gap
[1]	Traffic-aware autoscaling	Incorporates traffic patterns into HPA for Kubernetes	Lacks scalability for multi-cluster deployments in edge computing
[2]	LSTM-based autoscaling	Analyzes real-time and historical metrics to forecast resource demands, enabling proactive scaling	Limited integration with scheduling frameworks for task placement
[5]	Proactive autoscaling	Predicts workload demands using Bi-LSTM models for preemptive scaling	Limited focus on heterogeneous resource capacities in edge environments
[9]	Federated autoscaling	Leverages FL models to predict workload variations across distributed edge nodes	High communication overhead and synchronization challenges
[13]	ILP-based scheduling	Optimizes container task placement using ILP models	Limited adaptability to dynamic workloads in real-time systems
[14]	Real-time scheduling	Allocates tasks in KubeEdge environments, focusing on real-time edge task execution	Does not incorporate latency-awareness in multi-node edge deployments
[15]	Resource-aware scheduling	Proposes a taxonomy for scheduling algorithms with a focus on adaptive mechanisms	Absence of integration with predictive autoscaling mechanisms
[16]	Multi-objective scheduling	Employs multi-objective optimization to balance latency and resource usage in task scheduling	Requires enhanced scalability for large-scale edge systems

network conditions to ensure efficient processing and timely responses to user demands. By integrating these heuristic and taxonomy-based approaches, edge computing systems can achieve improved performance, adaptability, and resource management.

Energy efficiency is a critical concern in edge computing scheduling algorithms. In [18] a workload scheduling approach based on deep reinforcement learning (DRL) has been proposed to balance workloads, reduce service time, and decrease task failure rates in edge environments. This method utilizes deep Q-network (DQN) algorithms to address the complexities of workload scheduling, with the aim of improving virtual machine utilization and overall system performance. Article [19] introduces energy-efficient scheduling algorithms to minimize computational overhead while maintaining performance in edge environments.

In the realm of edge computing, DRL has been used to improve task scheduling efficiency. In [20] the DRL-based task scheduling algorithm has been introduced to intelligently manage tasks in edge computing settings, focusing on reducing service delay and traffic load. This approach uses reinforcement learning to optimize task assignments, thereby enhancing the efficiency of edge computing systems.

ILP-based approaches have also been used to optimize resource allocation. The authors of [21] use ILP models to optimize the placement of service function chains (SFC) in edge cloud environments, integrating workload predictions from LSTM models to improve efficiency. Similarly, paper [22] proposes the combined predictive autoscaler (COPA), which combines horizontal and vertical scaling to optimize resource usage in Kubernetes clusters.

Research is summarized in the Tab. 1 highlights significant advances in the fields of autoscaling and scheduling techniques.

3. ILP Formulation for Joint Autoscaling and Scheduling

To optimize resource allocation, task placement, and replica scaling simultaneously in the Kubernetes-based edge-cloud environment using a linear framework, we propose an ILP model. This model uses predictions from the LSTM model to make proactive decisions, with the aim of minimizing a composite cost function reflecting task priorities, network latency, and a linear cost for scaling, while adhering to node resource constraints. The decision variables are the following.

- $x_{p,n}$: binary decision variable, where $x_{p,n} = 1$ if the task p is assigned to node n and $x_{p,n} = 0$ otherwise. This represents the *scheduling* decision.
- R: non-negative integer decision variable representing the total target number of container replicas to be maintained across the cluster, as determined by the integrated optimization. This represents the *autoscaling* decision.

Parameters used in the proposed model:

- c_p: cost or inverse priority associated with placing task p. A higher c_p might represent a lower priority task or a higher intrinsic cost of running it.
- RTT_n: predicted round-trip time parameter for node n. In this work, RTT_n is defined as the predicted latency between edge node n and a designated central point within

the cluster, the Kubernetes API server. This serves as a proxy for the general responsiveness and accessibility of node n from a control or coordination perspective. This value is forecasted using a dedicated LSTM model.

The inputs to this LSTM model to predict RTT_n for a specific node n include its own historical RTT_n values (to the central point) from previous time intervals, the current CPU and memory utilization of node n, and the current network interface traffic statistics for node n (e.g., bytes in/out, packets in/out).

These input metrics are collected through Prometheus. The LSTM is trained offline on historical data to learn patterns and predict RTT_n for the subsequent operational interval. This predicted RTT_n is then fed as a constant parameter into each instance of the ILP optimization problem.

- γ: non-negative penalty factor for network latency. Controls the importance of minimizing latency during task placement.
- β: non-negative *linear* scaling cost factor. This weight represents the cost associated with the deployment and maintenance of each replica. Penalize solutions linearly on the basis of the total number of replicas R.
- CPU_p, memory_p: resource requirements (CPU cores, memory units) of a single instance/replica of task p.
- CPU_n, memory_n: resource capacities (available CPU cores, memory units) of edge node n.
- N: total number of edge nodes.
- P: total number of tasks to be scheduled.

The following constraints ensure valid scheduling and resource allocation. Each task must be assigned to exactly one node. To achieve this, the following formula is applied:

$$\sum_{n=1}^{N} x_{p,n} = 1, \quad \forall p \in \{1, \ldots, P\} . \tag{1}$$

The total CPU usage of tasks assigned to a node must not exceed the CPU capacity. To prevent over-commitment of CPU resources on any node, we use:

$$\sum_{p=1}^{P} x_{p,n} \cdot \mathrm{CPU}_p \leqslant \mathrm{CPU}_n, \quad \forall n \in \{1, \ldots, N\} . \tag{2}$$

Total memory usage of tasks assigned to a node must not exceed the memory capacity:

$$\sum_{p=1}^{P} x_{p,n} \cdot \mathrm{Memory}_p \leqslant \mathrm{Memory}_n, \quad \forall n \in \{1, \ldots, N\} . \tag{3}$$

The task-to-node assignment must be binary:

$$x_{p,n} \in \{0, 1\}, \quad \forall p \in \{1, \ldots, P\}, \ \ n \in \{1, \ldots, N\} . \tag{4}$$

The total number of replicas must be non-negative:

$$R \geqslant 0, \quad R \in \mathbb{Z} . \tag{5}$$

The goal is to minimize the combined linear cost of task placement, network latency, and replica scaling:

$$\min \sum_{p=1}^{P} \sum_{n=1}^{N} (c_p \cdot x_{p,n} + \gamma \cdot \mathrm{RTT}_n \cdot x_{p,n}) + \beta \cdot R . \tag{6}$$

The objective function (6) aims to find the optimal balance between scheduling efficiency and resource scaling costs using a purely linear formulation:

- The scheduling cost term $(c_p \cdot x_{p,n})$ accumulates the intrinsic cost c_p of placing the task p on node n. Favors placing low-cost (high-priority) tasks.
- The latency penalty term $(\gamma \cdot \mathrm{RTT}_n \cdot x_{p,n})$ adds a penalty proportional to the predicted latency (RTT_n) of node n where the task p is placed. The factor γ weights the importance of latency. Minimizing this drives tasks towards low-latency nodes.
- The linear scaling cost term $(\beta \cdot R)$ adds a cost that increases *linearly* with the total number of replicas R deployed. The factor β represents the cost per replica. This encourages resource efficiency by penalizing unnecessarily high replica counts.

The ILP solver finds the optimal integer values for $x_{p,n}$ (task placement) and R (total replicas) that minimize this linear objective function while satisfying all constraints (1)–(5).

The integration with LSTM remains the same as for ILP. The LSTM model provides predictive inputs:

- The predicted round-trip time RTT_n is used directly in the latency penalty term.
- Other LSTM predictions can inform the setting of parameters like c_p, CPU_p, or memory_p before solving the ILP.

By integrating autoscaling and scheduling equations into the ILP framework, the proposed model addresses the challenges of dynamic workload management in edge-cloud environments. The Gurobi solver is used to solve the ILP, ensuring efficient optimization of resources while maintaining low response times.

4. Proposed Methodology

This section details the methodology of the proposed system, covering system workflow, dataset preparation, scheduling algorithm, auto-scaling mechanism, and mathematical modeling using ILP for combined scheduling and autoscaling.

4.1. System Model

The system model, depicted in Fig. 1, integrates auto-scaling, scheduling, and predictive modeling to achieve efficient resource management in Kubernetes-based edge-cloud environments. At the core of the design depicted in Fig. 2 is a seamless workflow that begins with collecting real-time resource metrics, such as CPU usage, memory consumption, and RTT from all nodes within the cluster.

Metric server as the monitoring agent, providing an uninterrupted stream of data essential for decision-making processes.

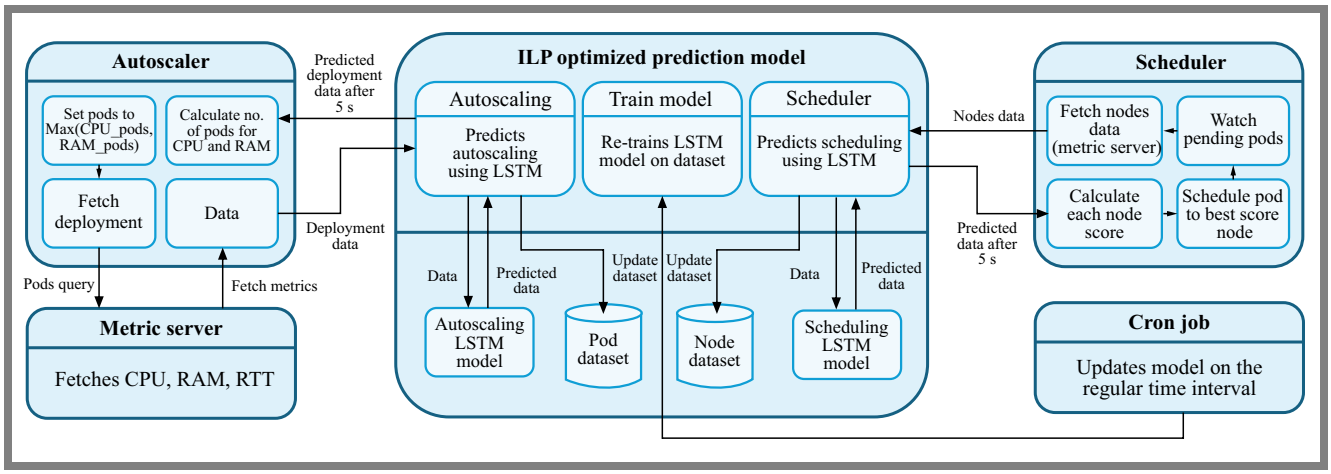

Fig. 2. Detailed system model.

The autoscaler processes these metrics and prepares them to forecast future workloads for 5 s using an LSTM-based prediction model. This predictive model utilizes historical and real-time data to estimate future resource requirements, including CPU and memory usage, network delays, and application request rates. Predicted values are stored alongside the actual metrics, creating a continuously updated dataset to retrain the LSTM model. This approach ensures that the accuracy of the prediction evolves with changes in workload patterns, making the system highly adaptive.

When resource demands exceed predefined thresholds, the autoscaler proactively scales resources by either increasing replicas or reallocating workloads to mitigate performance bottlenecks. Similarly, the scheduler evaluates all nodes within the cluster to determine the most suitable deployment location for new or pending pods. This evaluation is based on a scoring mechanism that incorporates predicted CPU and memory usage, task priority, pod affinity, and network parameters such as RTT. The node with the highest score is selected for pod deployment, ensuring efficient and balanced resource allocation.

To further enhance system performance, the model is designed to retrain the LSTM predictor at regular intervals using a cron job. The updated data set collected during operations allows the retraining process to adapt to evolving workload behaviors, ensuring the system remains capable of handling unpredictable and dynamic resource demands.

4.2. Dataset

The data set used in this investigation was created using Prometheus, a powerful open-source monitoring tool. Prometheus collects real-time metrics related to resource usage from Kubernetes-based edge-cloud environments. The dataset consists of two main components: pod-level data for autoscaling and node-level data for scheduling. Each data set contains features that help predict future resource utilization and optimize resource allocation strategies.

The pod-level dataset focuses on metrics crucial to making auto-scaling decisions. The data set includes information on CPU usage, memory utilization, RTT, and future predictions. The data is continuously monitored and collected by Prometheus. A detailed description of the pod-level dataset features is provided in the Tab. 2.

Tab. 2. Description of the dataset at the pod level.

Feature	Description
timestamp	The timestamp indicating the time of the measurement
cpu	CPU usage of the pod
memory	Memory usage of the pod
rtt	Round-trip time or latency for the pod
next_rtt	Round-trip time or latency 5 s after the current time
next_memory	Memory usage 5 s after the current time
next_cpu	CPU usage 5 s after the current time

Tab. 3. Description of the node-level dataset.

Feature	Description
timestamp	The timestamp indicating the time of the measurement
nodename	The name of the node
cpu	CPU usage of the node
memory	Memory usage of the node
rtt	Round-trip time or latency for the node
next_rtt	Round-trip time or latency 5 s after the current time
next_memory	Memory usage 5 s after the current time
next_cpu	CPU usage 5 s after the current time

The node-level data set is designed to help schedule pods to the most optimal nodes. It includes both current and predicted metrics for nodes such as CPU usage, memory usage, and RTT. This data set is essential to implement an efficient

scheduling algorithm. Table 3 provides a detailed description of the features of the node-level dataset.

4.3. The Algorithm Description

The proposed methodology, shown in Algorithm 1, adopts a cyclical, multiphase strategy to integrate predictive insights with optimization for resource management in Kubernetes-based edge environments. This structure aims at both proactive adjustments and refined decision-making. The process involves the following phases:

1) Proactive autoscaling (lines 11–23). This phase leverages LSTM predictions (PredictMetricValues), derived from current Prometheus metrics, to forecast future resource demands. Based on these predictions versus operational thresholds, a preliminary target replica count (NoofReplicasHeuristic) is calculated heuristically (lines 20–21). An initial scaling action (ScaleDeployment) might optionally be performed at this stage (line 22) if the heuristic count differs significantly from the current state, allowing a rapid response to predicted load shifts.
2) Heuristic scheduling (lines 24–29). Following autoscaling considerations, this phase performs a quick heuristic placement for pending tasks. Calculate a score $(Score_n)$ for each node using the function f, based on current or predicted resources and latency RTT_n (line 26). Each task is then preliminarily assigned to the node identified as having the best score (BestNode) (lines 28–29).
3) ILP optimization and refinement (lines 30–39): The final phase employs ILP for comprehensive optimization. An ILP problem is formulated using the revised model, i.e. constraints (1)–(4) and objective (6) (lines 32–33). The inputs include node states, task requirements, predicted RTT_n by LSTM, and tuning parameters γ, β, c_p. The ILP is solved using Gurobi (line 34) to determine both the final optimal task placement $x_{p,n}$ and the final optimal total replica count $R_{optimal}$. This ILP solution refines the decisions of the previous phases. The resulting $R_{optimal}$ dictates the definitive scaling action (lines 36–37), and the optimized $x_{p,n}$ determines the final task deployment (line 39).

This multiphase design allows the system to potentially react quickly using predictive heuristics (phases 1 and 2) while leveraging the comprehensive optimization power of ILP (phase 3) for refinement and determining the definitive scaling and placement actions. Practical considerations such as prediction accuracy, interphase delays, and ILP solve time remain relevant for real-world performance.

4.4. Cron Job for LSTM Retraining

To maintain prediction accuracy, the LSTM model is periodically re-trained using updated datasets. A cron job is implemented to automate this process. The steps involved are as follows:

- Fetch updated metrics (CPU, memory, RTT) from Prometheus,
- Update the training dataset with the latest metrics,
- Retrain the LSTM model with the updated dataset to improve prediction accuracy,
- Deploy the updated model into the system for future predictions.

Tab. 4. Hardware and software configuration.

Component	Specification
Processor	Intel core i5 7th Gen
RAM	128 GB DDR4
Storage	30 GB SSD per node
Operating system	Ubuntu 20.04 LTS
Kubernetes version	1.29
KubeEdge version	1.19
Nodes	11: 1 cloud node, 10 edge nodes
RAM per node	4 GB
Virtualization	VMware workstation
Orchestration tool	Kubectl
Programming framework	TensorFlow, Python
Deployed application	Web App using Ngnix

Tab. 5. Configuration of the LSTM model.

Parameter	Value
Number of LSTM layers	16
Number of epochs	50
Batch size	64
Optimizer	Adam
Loss function	Categorical cross entropy

5. Results and Discussion

The data set was generated in a multinode KubeEdge setup, consisting of one cloud node and ten edge nodes, and metrics were collected at both node and pod levels using Prometheus and bash scripts over a 10-hour period, capturing diverse workloads and resource utilization patterns. Table 4 summarizes the configuration used, while the LSTM model configuration is summarized in Table 5. Workloads to evaluate autoscaling mechanisms were generated using the Apache Benchmark (ab) tool, which aims at the deployed web server application.

For the evaluations focusing on fixed sustained load for response time, CPU/memory utilization), a total of 100 000 HTTPS GET requests ab Apache tool with concurrency of 100. This configuration creates a *closed-loop workload model*. In this model, ab attempts to maintain 100 concurrent active connections to the server. As soon as a request receives a response, ab immediately issues a new request on

Algorithm 1 ILP optimized LSTM-based autoscaling and scheduling.

Input:
  Historical metrics $\{CPU_t, MEM_t, RTT_t\}_{t=1}^{T}$ from metric server
  Current replica count $R_{current}$ from Kubernetes
  Threshold values for CPU, memory, and RTT: $\mathrm{Threshold}_{\mathrm{CPU}}, \mathrm{Threshold}_{\mathrm{memory}}, \mathrm{Threshold}_{\mathrm{RTT}}$
  Resource capacities limit of nodes: $\mathrm{CPU}_n, \mathrm{Memory}_n$, and RTT_n
  Tasks and their resource requirements: $\{CPU_p, MEM_p\}_{p=1}^{P}$
Output:
  Final optimized replica count $R_{optimal}$ deployed
  Task-to-node mapping $x_{p,n}$ used for deployment
while true **do**
  Phase 1. Autoscaling
  DesiredReplicaHeuristic ← 1
  FetchedMetrics ← FetchMetricsFromPrometheus()
  CurrentReplicas ← GetCurrentReplicas(Deployment)
  for metric in FetchedMetrics **do**
    PredictedMetricValue ← PredictMetricValues(metric)
    DesiredMetricValue ← GetDesiredMetricValue(metric)
    MetricDesiredReplicas ← ⌈CurrentReplicas · (PredictedMetricValue / DesiredMetricValue)⌉
    DesiredReplicaHeuristic ← max(DesiredReplicaHeuristic, MetricDesiredReplicas)
  end for
  NoofReplicasHeuristic ← DesiredReplicaHeuristic
  if NoofReplicasHeuristic ≠ CurrentReplicas **then**
  end if
  Phase 2. Scheduling
  For each task p calculate node scores using resource metrics:
    $Score_n \leftarrow f(\mathrm{CPU}_n, \mathrm{Memory}_n, \mathrm{RTT}_n)$
  Identify the most suitable node for each task:
    $BestNode \leftarrow \arg\max_n(Score_n)$
  Schedule tasks to nodes based on the highest scores
  Phase 3. ILP Optimization
  Define decision variables: $x_{p,n}$ (binary), R_{optimal} (integer)
  Formulate constraints using Eqs. (1)–(4)
  Reference objective function from Eq. (6)
  Solve ILP using Gurobi solver to determine optimal $x_{p,n}$ and optimal replica count R_{optimal}
    ▷ The solved R_{optimal} refines/overrides $NoofReplicasHeuristic$ from phase 1
  if R_{optimal} ≠ CurrentReplicas **then**
    ScaleDeployment(R_{optimal})
  end if
  Deploy tasks to nodes based on the optimized $x_{p,n}$
end while

that connection, continuing until 100 000 total requests are completed. The request arrival process throughput measurements obtained using ab tool are based on a deterministic arrival process, where new requests are initiated as soon as existing ones complete, constrained by the specified concurrency level. This approach subjects the system to continuous high stress.

The Apache Benchmark (ab) tool was used with different total request counts (-n parameter), specifically 10 000, 30 000, 50 000, 75 000, and 100 000 requests. The concurrency level was kept constant at 100 (-c 100) for all of these runs. This variation in the total number of requests, while maintaining the same concurrency, effectively changes the duration of the sustained load test. The nature of the request generation remained a closed-loop model (100 concurrent connections sending requests as fast as the server responds), allowing for the measurement of sustained throughput under different total work performed.

For the context of these experiments, throughput is defined as the average rate at which the system successfully processes requests over the entire duration of a given test run. It is calculated as follows:

$$\text{Throughput} = \frac{\text{Total successfully completed requests}}{\text{Total time taken for the test run [s]}} . \quad (7)$$

This value is directly reported by the ab tool upon completion of each test. During the experiments, it was ensured that the server did not explicitly reject requests due to overload; thus, the measured throughput primarily reflects the sustained

Tab. 6. Evaluation metrics for scheduling predictions.

Model	Evaluation metric	CPU	Memory
LSTM	MSE	0.28	0.45
	MAE	0.22	0.39

Tab. 7. Optimized LSTM predictions with ILP.

Evaluation metric	CPU	Memory
MSE	0.166	0.304
MAE	0.210	0.445

Tab. 8. Evaluation metrics for autoscaling predictions.

Model	Evaluation metric	CPU	Memory
LSTM	MSE	0.156	0.104
	MAE	0.247	0.195

service capacity of the autoscaled application deployment rather than being affected by significant request loss.

5.1. Scheduling Predictions (Nodes)

Evaluation of models to predict CPU and memory usage revealed that the LSTM model provides highly accurate predictions. The evaluation metrics for the CPU and memory usage predictions are presented in Table 6. One may notice that the LSTM model achieved an MSE of 0.28 (CPU) and 0.45 (memory), with MAE values of 0.22 (CPU) and 0.39 (memory).

Table 7 demonstrates the impact of optimization using ILP. The MSE and MAE values for CPU and memory predictions are reduced. This improvement is achieved because ILP minimizes the prediction error by systematically adjusting the task allocations, ensuring greater accuracy.

5.2. Autoscaling Prediction (Pods)

The LSTM model is also employed to predict CPU and memory usage for the auto-scaling mechanism. The evaluation metrics for autoscaling predictions are detailed in the Tab. 8. The values obtained show the performance of the LSTM model for auto-scaling predictions across CPU and memory usage. The model achieved a mean squared error (MSE) of 0.156 for the CPU and 0.104 for memory, indicating high precision in predicting resource requirements. Similarly, the mean absolute error (MAE) values were 0.247 for the CPU and 0.195 for memory, demonstrating the reliability in minimizing prediction deviations.

Table 9 highlights the improved prediction metrics after optimizing LSTM predictions with ILP. By reducing errors in resource estimation, the system achieves a better alignment between predicted and actual usage, leading to improved resource allocation efficiency.

Tab. 9. Optimized LSTM predictions with ILP for autoscaling.

Evaluation metric	CPU	Memory
MSE	0.135	0.089
MAE	0.356	0.202

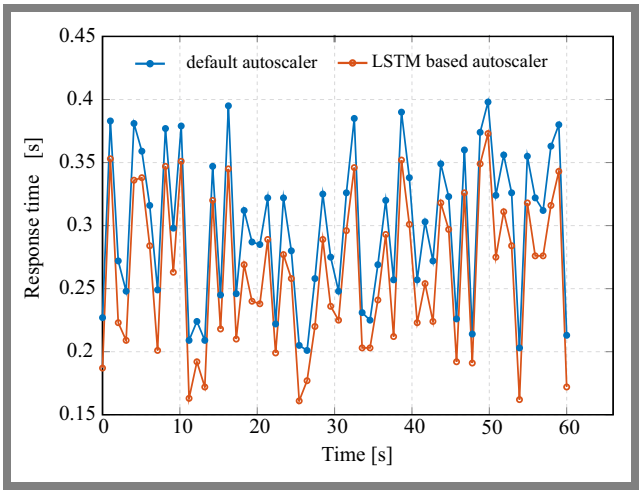

Fig. 3. Response time comparison: HPA vs. LSTM autoscaling.

5.3. Autoscaling Results

The evaluation of CPU and memory utilization, as well as response time, was performed under a fixed workload of 50 000 HTTPS requests using the ab tool with 100 concurrent requests at a time. This workload level was selected to simulate medium to high resource utilization scenarios, providing insight into system behavior under significant demand. The metrics obtained offer a detailed view of resource consumption and autoscaling efficiency under real conditions.

Response time refers to the end-to-end duration measured by the Apache Benchmark client for each individual HTTPS request, capturing the total time from request initiation to the reception of the complete response from the application server. This metric reflects the perceived latency under the applied load. This measured application response time should be distinguished from the predicted network latency parameter RTT_n used within the ILP objective function, which serves as an internal factor optimized to influence this overall externally measured response time.

Figure 3 presents a comparative analysis of the response times between the default autoscaling mechanisms over time. The graph reveals that the LSTM-based autoscaler consistently maintains lower response times throughout the observation period, with values ranging between 0.15 and 0.5 s. The LSTM model demonstrates superior performance by achieving a 12.5% reduction in response time compared to the default autoscaler.

The default autoscaler exhibits more pronounced fluctuations and generally higher response times, with peaks reaching approximately 0.4 s. In contrast, the LSTM-based autoscaler maintains more stable performance with an average response time of 0.262 s, compared to 0.298 s. This improvement can be attributed to the ability of the LSTM model to predict

Tab. 10. Response time metrics: HPA vs. LSTM autoscaling.

Metric	HPA RT	LSTM autoscaling RT
Minimum	0.201 s	0.161 s
Average	0.298 s	0.262 s
Maximum	0.398 s	0.373 s

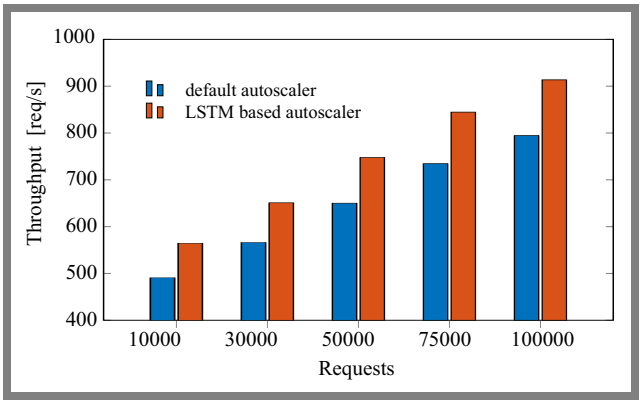

Fig. 4. Throughput comparison: HPA vs. LSTM autoscaling.

resource requirements and proactively adjust allocations, resulting in more efficient resource utilization and reduced latency.

The temporal pattern shows that while both autoscalers experience periodic fluctuations in response times, the LSTM-based approach maintains better consistency and lower overall latency. This improved stability and reduced response time demonstrate the effectiveness of the LSTM model in dynamic resource allocation, particularly in handling varying workload conditions in edge cloud environments.

Table 10 highlights the statistical metrics for response time. LSTM autoscaling consistently achieves better minimum, average, and maximum response times compared to HPA, ensuring faster response to workload fluctuations.

Figure 4 presents a comparison of throughput performance between the LSTM-based autoscaler and the default autoscaler (HPA) across varying request loads from 10 000 to 100 000 requests. The LSTM autoscaler consistently demonstrates superior performance, achieving a 29.2% improvement in throughput compared to HPA. The graph illustrates the progressive increase in throughput as the request volume increases. The LSTM autoscaler maintains a steeper growth trajectory, starting at approximately 564.52 req/sec at 10 000 requests and reaching 913.54 req/s at 100 000 requests. On the contrary, the default autoscaler shows a more modest progression from 490.89 req/s to 794.39 req/s over the same range.

Figure 4 also provides a visualization of the performance gap between the two approaches at specific request intervals. The LSTM autoscaler achieves an average throughput of 744.36 req/s, outperforming the default autoscaler's 647.18 req/s. This performance differential becomes more pronounced with higher request volumes, demonstrating superior capability in handling increased workload demands.

Tab. 11. Summary of performance metrics.

Metric	HPA	LSTM autoscaling
Minimum	490.888 req/s	564.521 req/s
Average	647.169 req/s	744.644 req/s
Maximum	794.385 req/s	913.543 req/s

This improvement in performance can be attributed to the ability of the LSTM model to predict resource requirements and proactively adjust scaling decisions, resulting in more efficient resource utilization and better handling of varying workload patterns in edge cloud environments.

Table 11 presents a summary of the performance for HPA and LSTM autoscaling. The LSTM autoscaling model consistently delivers higher throughput, with a minimum of 564.52 req/s, exceeding HPA's 490.89 req/s.

The average throughput for LSTM autoscaling is 744.644 req/s, while HPA achieves 647.169 req/s, indicating an improvement. At peak load, LSTM autoscaling reaches a maximum throughput of 913.543 req/s compared to HPA's 794.385 req/s. These results highlight the scalability and efficiency of LSTM-based autoscaling over HPA.

5.4. Combined Autoscaling and Scheduling Results

Integrating LSTM-based autoscaling and scheduling ensures a coordinated approach, improving workload distribution and resource management. Evaluation of CPU and memory utilization, as well as response time, was performed using a fixed workload of 50 000 requests generated with the Apache Benchmark (ab) tool. This workload level was selected to represent a realistic medium-load scenario, providing a comprehensive view of system performance under consistent demand.

Figure 5 illustrates the response time comparison between LSTM autoscaling with default scheduler and LSTM-based autoscaling and scheduling over time. The LSTM autoscaling with default scheduler exhibits higher response times throughout the observation period, fluctuating between 0.201 and 0.397 s. The response time pattern shows notable variations, particularly between the 2040 s interval, indicating less stable performance. On the contrary, the LSTM-based autoscaling and scheduling approach demonstrates consistently lower response times across the entire timeline.

This combined approach maintains response times between 0.158 and 0.368 s, achieving a 12% improvement in the median response time (0.259 s vs. 0.294 s). The graph shows more stable performance with fewer fluctuations, particularly evident in the 30–50 s range, where the response time variations are notably smaller than the default approach.

The improved stability and lower response times can be attributed to the combined effect of auto-scaling with scheduling. The integrated approach demonstrates superior resource allocation efficiency, with the LSTM models working in tandem to predict resource requirements and optimize pod placement. This coordinated decision-making results in more predictable performance patterns and reduced latency, as ev-

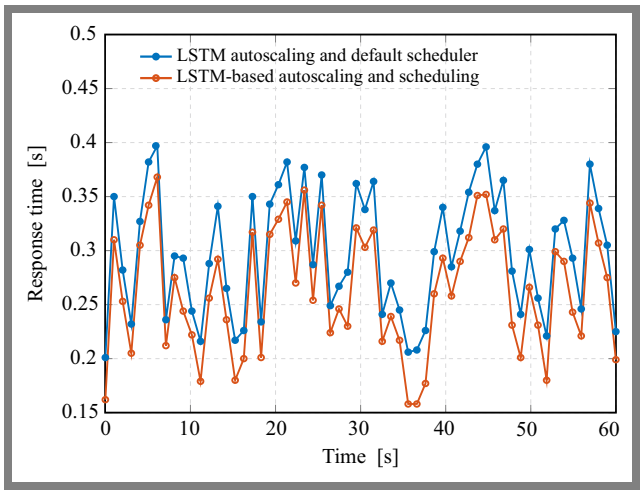

Fig. 5. Response time vs. time for combined scheduling and autoscaling.

Tab. 12. Comparison of response time for combined scheduling and autoscaling.

Metric	LSTM based autoscaling with default scheduler	LSTM autoscaling and scheduling
Minimum	0.201 s	0.158 s
Average	0.294 s	0.259 s
Maximum	0.397 s	0.368 s

idenced by the 21.4% improvement in minimum response time (from 0.201 to 0.158 s) and the 7.3% reduction in maximum response time (from 0.397 to 0.368 s). Performance improvements in all metrics underscore the effectiveness of the combined LSTM-based approach in maintaining optimal system responsiveness under varying workload conditions.

Figure 6 presents a comparison of throughput performance between LSTM autoscaling with the default Kubernetes scheduler and the combined LSTM-based autoscaling and scheduling approach. The analysis reveals several significant performance patterns across varying request loads. At the lower end of the request spectrum (10 000 requests), the combined LSTM-based approach demonstrates an initial throughput advantage of 628.14 req/s compared to 611.33 req/s for the default scheduler, that is, a 2.75% improvement. This performance gap widens with load level.

The throughput enhancement becomes pronounced at higher request volumes, reaching 930.42 req/s vs. 907.78 req/s at 100 000 requests, maintaining a 2.49% performance gain. The system shows excellent scalability, with both approaches maintaining near-linear throughput growth from 10 000 to 100 000 requests. The LSTM-based combined approach maintains a consistent performance improvement of average 2.61% at all test points. At medium load (50 000 requests), the LSTM-based system processes 758.99 req/s compared to 739.73 req/s for the default scheduler, demonstrating robust performance under typical operating conditions. The highest absolute performance gain is observed at 100 000 requests, where the LSTM-based system processes an additional 22.64

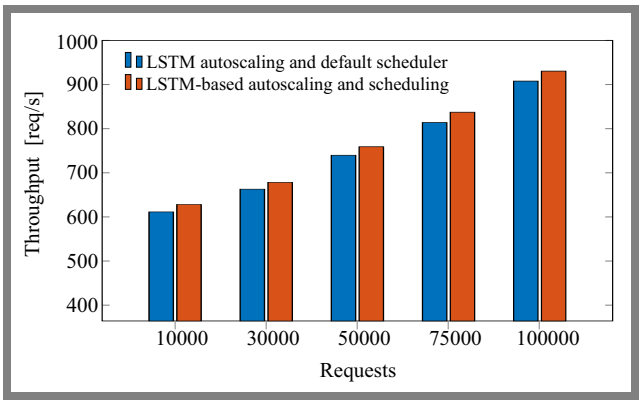

Fig. 6. Comparison of performance for combined scheduling and autoscaling.

Tab. 13. Summary of performance metrics: HPA vs. LSTM autoscaling.

Metric	HPA	LSTM autoscaling
Minimum	611.331 req/s	628.14 req/s
Average	747.0824 req/s	766.217 req/s
Maximum	907.778 req/s	930.416 req/s

req/s. This sustained performance improvement across all request levels demonstrates the effectiveness of integrating LSTM-based decision-making in both the scheduling and autoscaling components. Table 13 summarizes the throughput performance of HPA and LSTM autoscaling on different request loads. LSTM-based autoscaling consistently outperforms HPA in all scenarios.

5.5. Combined Autoscaling and Scheduling with and without ILP

Here a comparative analysis of the combined autoscaling and scheduling approach with and without ILP optimization is provided. The evaluation focuses on response time as the key performance metric and data collected under a fixed workload of 50 000 HTTPS requests using the ab tool. This workload level provides a realistic scenario to assess the efficiency of ILP optimization in minimizing latency.

The response time analysis reveals distinct performance characteristics between the two approaches (Fig. 7). The combined autoscaling and scheduling without ILP exhibits response times fluctuating between 0.21 and 0.40 s, with variations particularly in the 20–30 s interval. In contrast, the ILP-enhanced approach demonstrates superior stability, maintaining response times between 0.15 and 0.33 s with reduced variance.

The throughput comparison across varying request loads shows consistent performance advantages for the LSTM-based approach. Starting at 10 000 requests, it achieves 628.14 req/s compared to 611.33 req/s for the default scheduler, i.e. a 2.75% improvement. This performance differential persists through higher loads, reaching 930.42 req/s versus 907.78 req/s at 100 000 requests.

The system demonstrates excellent scalability with near-linear throughput growth throughout the test range, maintaining

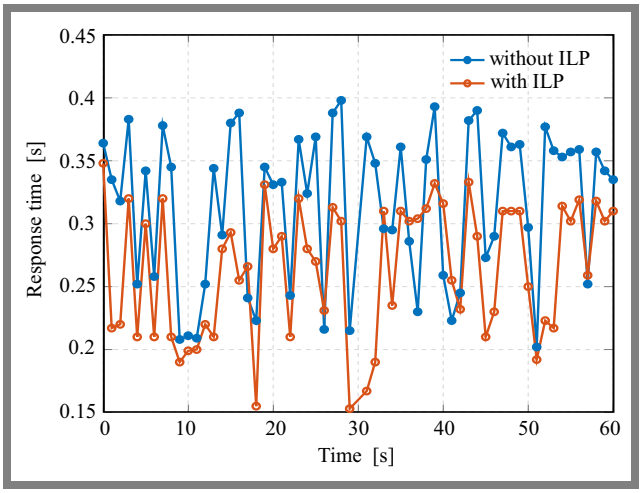

Fig. 7. Comparison of response times for combined autoscaling and scheduling with and without ILP.

Tab. 14. Comparison of response time for combined scheduling and autoscaling.

Metric	LSTM autoscaling and scheduling	ILP-enhanced LSTM autoscaling and scheduling
Minimum	0.210 s	0.152 s
Median	0.315 s	0.262 s
Maximum	0.397 s	0.348 s

an average improvement of 2.61% across all test points. At medium load (50 000 requests), the LSTM-based system processes 758.99 req/s compared to 739.73 req/s for the default scheduler, while the highest absolute performance gain is observed at 100 000 requests with an additional 22.64 req/s. The impact of ILP optimization is particularly evident in maintaining more consistent performance levels, especially during the 40–60 s period, where it stabilizes around 0.30 s, demonstrating enhanced efficiency in resource allocation and workload distribution.

The metrics in the Tab. 14 highlight the improvements achieved by the ILP-enhanced LSTM autoscaling and scheduling approach. Compared to the non-ILP method, the median response time was reduced from 0.315 to 0.263 s, demonstrating enhanced efficiency, particularly in handling dynamic workloads with lower latency.

Figure 8 illustrates an analysis of throughput performance between the ILP-enhanced and standard LSTM approaches for combined autoscaling and scheduling. The ILP-enhanced solution demonstrates superior performance across all request volumes, with the average throughput increasing from 756.34 req/s to 815.50 req/s, representing a 7.8% improvement.

The performance advantage becomes more pronounced under higher workloads, with maximum throughput reaching 1 010.75 req/s compared to 929.68 req/s in the non-ILP approach, showing an 8.7% increase. Even at lower request volumes, the ILP-enhanced method maintains better efficiency, with minimum throughput improving from 581.02 req/s

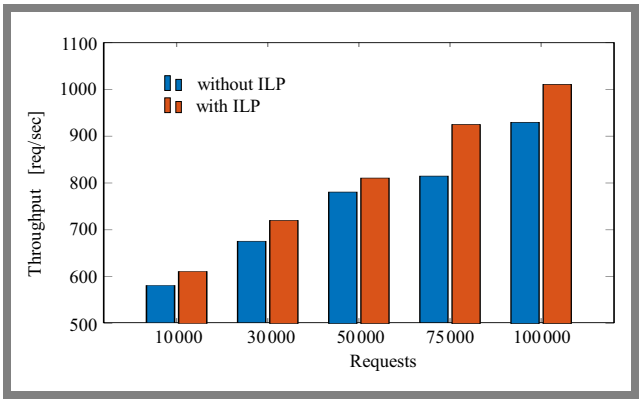

Fig. 8. Performance comparison for combined LSTM autoscaling and scheduling with and without ILP.

Tab. 15. Performance metrics: combined LSTM autoscaling and scheduling vs. ILP-enhanced approach.

Metric	LSTM autoscaling and scheduling	ILP-enhanced LSTM autoscaling and scheduling
Minimum	581.02 req/s	610.88 req/s
Average	756.34 req/s	815.50 req/s
Maximum	929.68 req/s	1010.75 req/s

to 610.88 req/s. The graph highlights consistent performance gains across all request volumes, particularly in the 75 000–100 000 request range, where the system demonstrates optimal resource utilization and workload management capabilities. Table 15 provides detailed comparison of throughput metrics.

6. Conclusions

This study addresses the limitations of default Kubernetes resource management by proposing an integrated framework that combines LSTM-based autoscaling and scheduling with ILP-based optimization. Using predictive modeling in conjunction with intelligent resource allocation, the ILP and LSTM-based system improves overall efficiency. A comparative evaluation between the LSTM-based autoscaling and scheduling system and the ILP- and LSTM-based system demonstrated a 12.34% reduction in response time and a 7.85% increase in throughput.

Despite the improvement in results, several limitations must be considered. The experiments were conducted in a virtualized environment using a stateless Web application. While efforts were made to approximate real-world conditions, the controlled nature of the testbed may not fully reflect the complexities encountered in practical deployments. Furthermore, the LSTM model was specifically trained and fine-tuned for the given use case. Applying the framework to other applications would likely require retraining the model with domain-specific historical data and adjusting parameters to suit different system dynamics.

References

[1] 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).

[2] S.T. Singh, M. Tiwari, and A.S. Dhar, "Machine Learning based Workload Prediction for Auto-scaling Cloud Applications", *2022 OPJU International Technology Conference on Emerging Technologies for Sustainable Development (OTCON)*, Raigarh, India, 2023 (https://doi.org/10.1109/OTCON56053.2023.10114033).

[3] I. Ahmad, M.G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container Scheduling Techniques: A Survey and Assessment", *Journal of King Saud University – Computer and Information Sciences*, vol. 34, pp. 3934–3947, 2022 (https://doi.org/10.1016/j.jksuci.2021.03.002).

[4] K. Senjab, S. Abbas, N. Ahmed, and A.R. Khan, "A Survey of Kubernetes Scheduling Algorithms", *Journal of Cloud Computing*, vol. 12, art. no. 87, 2023 (https://doi.org/10.1186/s13677-023-00471-1).

[5] K. Zhu *et al.*, "Proactive Hybrid Autoscaling for Container-Based Edge Applications in Kubernetes", *Lecture Notes of the Institute for Computer Sciences*, vol. 574, pp. 330–345, 2024 (https://doi.org/10.1007/978-3-031-65123-6_24).

[6] Z. Wang, Q. Zhu, and Y. Hou, "Multiworkflow Scheduling in Edge-cloud Computing by African Vulture Optimization Algorithm", *2024 11th International Forum on Electrical Engineering and Automation (IFEEA)*, Shenzhen, China, 2024 (https://doi.org/10.1109/IFEEA64237.2024.10878706).

[7] J. Li, P. Singh, and S. Toor, "Proactive Autoscaling for Edge Computing Systems with Kubernetes", *Proc. of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC'21)*, art. no. 22, pp. 1–8, 2022 (https://doi.org/10.1145/3492323.3495588).

[8] L.H. Phuc *et al.*, "Node-based Horizontal Pod Autoscaler in KubeEdge-based Edge Computing Infrastructure", *IEEE Access*, vol. 10, pp. 134417–134426, 2022 (https://doi.org/10.1109/ACCESS.2022.3232131).

[9] J. Dogani and F. Khunjush, "Proactive Auto-scaling Technique for Web Applications in Container-based Edge Computing Using Federated Learning Model", *Journal of Parallel and Distributed Computing*, vol. 187, art. no. 104837, 2024 (https://doi.org/10.1016/j.jpdc.2024.104837).

[10] T.-X. Do and V.K.N. Tan "Hybrid Autoscaling Strategy on Container-Based Cloud Platform", International Journal of Software Innovation, vol. 10, 2022, pp. 1–12. (https://doi.org/10.4018/IJSI.292019).

[11] X. Feng *et al.*, "Adaptive Container Auto-scaling for Fluctuating Workloads in Cloud", *Future Generation Computer Systems*, vol. 172, art. no. 107872, 2025 (https://doi.org/10.1016/j.future.2025.107872).

[12] S.D. Konidena, "Efficient Resource Allocation in Kubernetes Using Machine Learning", *International Journal of Innovative Science and Research Technology*, vol. 9, 2024 (https://doi.org/10.38124/ijisrt/IJISRT24JUL607).

[13] J. Zhou, S. Pal, C. Dong, and K. Wang, "Enhancing Quality of Service Through Federated Learning in Edge-cloud Architecture", *Ad Hoc Networks*, vol. 156, art. no. 103430, 2024 (https://doi.org/10.1016/j.adhoc.2024.103430).

[14] S.-H. Kim and T. Kim, "Local Scheduling in KubeEdge-based Edge Computing Environment", *Sensors*, vol. 23, art. no. 1522, 2023 (https://doi.org/10.3390/s23031522).

[15] M. Raeisi-Varzaneh, O. Dakkak, A. Habbal, and B.-S. Kim, "Resource Scheduling in Edge Computing: Architecture, Taxonomy, Open Issues and Future Research Directions", *IEEE Access*, vol. 11, pp. 25329–25350, 2023 (https://doi.org/10.1109/ACCESS.2023.3256522).

[16] Z. Shi and Z. Shi, "Multi-node Task Scheduling Algorithm for Edge Computing Based on Multi-Objective Optimization", *Journal of Physics: Conference Series*, vol. 1607, art. no. 012017, 2020 (https://doi.org/10.1088/1742-6596/1607/1/012017).

[17] K. Wang *et al.*, "Computing Aware Scheduling in Mobile Edge Computing System", *Wireless Networks*, vol. 27, pp. 4229–4245, 2021 (https://doi.org/10.1007/s11276-018-1892-z).

[18] G. Vijayasekaran and M. Duraipandian, "Resource Scheduling in Edge Computing IoT networks Using Hybrid Deep Learning Algorithm", *System Research and Information Technologies*, pp. 86–101, 2022 (https://doi.org/10.20535/SRIT.2308-8893.2022.3.06).

[19] Y. Lu *et al.*, "EA-DFPSO: An Intelligent Energy-efficient Scheduling Algorithm for Mobile Edge Networks", *Digital Communications and Networks*, vol. 8, pp. 237–246, 2022 (https://doi.org/10.1016/j.dcan.2021.09.011).

[20] P. Khoshvaght *et al.*, "A Multi-objective Deep Reinforcement Learning Algorithm for Spatio-temporal Latency Optimization in Mobile IoT-enabled Edge Computing Networks", *Simulation Modelling Practice and Theory*, vol. 143, art. no. 103161, 2025 (https://doi.org/10.1016/j.simpat.2025.103161).

[21] P. Vishesh *et al.*, "Optimized Placement of Service Function Chains in Edge Cloud with LSTM and ILP", *SN Computer Science*, vol. 6, art. no. 44, 2024 (https://doi.org/10.1007/s42979-024-03539-0).

[22] Z. Ding and Q. Huang, "COPA: A Combined Autoscaling Method for Kubernetes", *2021 IEEE International Conference on Web Services (ICWS)*, Chicago, USA, 2021 (https://doi.org/10.1109/ICWS53863.2021.00061).

Shivan Singh, B.Eng.
School of Computer Science and Engineering
https://orcid.org/0009-0004-7894-3858
E-mail: 01fe21bcs246@kletech.ac.in
KLE Technological University, Hubballi, Karnataka, India
https://www.kletech.ac.in

Narayan D.G., Ph.D.
School of Computer Science and Engineering
https://orcid.org/0000-0002-2843-8931
E-mail: narayan_dg@kletech.ac.in
KLE Technological University, Hubballi, Karnataka, India
https://www.kletech.ac.in

Sadaf Mujawar, M.Tech.
Department of Computer Science and Engineering
https://orcid.org/0009-0007-5434-0114
E-mail: sadaf.savanur@kletech.ac.in
KLE Technological University, Hubballi, Karnataka, India
https://www.kletech.ac.in

G.S. Hanchinamani, Ph.D.
Department of Computer Science and Engineering
https://orcid.org/0000-0002-8791-0351
E-mail: gs_hanchinamani@kletech.ac.in
KLE Technological University, Hubballi, Karnataka, India
https://www.kletech.ac.in

P.S. Hiremath, Ph.D.
Department of MCA
https://orcid.org/0000-0001-7640-6937
E-mail: pshiremath@kletech.ac.in
KLE Technological University, Hubballi, Karnataka, India
https://www.kletech.ac.in