Intelligent Autoscaling with Attention-based Reinforcement Learning for SLA-Aware Resource Management in Edge-Cloud Environments

Faraz Shaikh, Gianluca Reali, Mauro Femminella
Department of Engineering, CNIT Research Unit
University of Perugia, Perugia, Italy
Email: faraz@dottorandi.unipg.it, gianluca.reali@unipg.it, mauro.femminella@unipg.it

***Abstract*—Modern cloud and edge computing infrastructures operate under rigorous Service Level Agreements (SLAs), which define Quality of Service (QoS) standards and key performance indicators (KPIs) such as latency, resource efficiency, and reliability. Meeting these requirements is increasingly challenging due to highly variable, bursty, and latency-sensitive workloads demanding intelligent, efficient, and adaptive resource management. To address these evolving needs, we introduce a novel autoscaling framework that combines the Proximal Policy Optimization (PPO) algorithm with a double-stacked Long Short-Term Memory (LSTM) network and an attention mechanism, enabling coordinated and proactive autoscaling in Kubernetes-based environments. The proposed approach utilizes a multidimensional discrete action space to simultaneously tune the Horizontal Pod Autoscaler (HPA) CPU targets, throughput multipliers, learning rate schedules, and policy enhancements, thereby effectively adapting to both short-term and long-term fluctuations in network traffic. Additionally, a forecasting module integrates LSTM-driven workload predictions with recent observation trends, allowing the autoscaler to predict dynamic changes and accommodate resources accordingly. Comprehensive simulations using real Azure Functions invocation traces from production environments justify that our system consistently maintains SLA, QoS, and KPI objectives, while delivering operational stability and enhanced resource efficiency for cloud-edge networks. Under strict SLA constraints, our proposed methodology achieves an average latency reduction of 87.5% and a median latency reduction of 71.5% compared to the baseline.**

***Index Terms*—autoscaling, kubernetes, LSTM, PPO, edge, cloud**

I. Introduction

The increasing trend in cloud-native, edge, and serverless computing has revolutionized the way modern applications are developed, deployed, and scaled. Edge computing refers to the use of computational resources located near the user [1]. Function-as-a-Service (FaaS) platforms such as Google Cloud Functions and Microsoft Azure Functions now serve as a basic building block of the digital infrastructure, abstracting server management and offering developers a seamless on-demand scaling [2]. However, under this abstraction lies a complex orchestration problem, i.e., to allocate computing resources in response to highly variable and often unpredictable workloads dynamically. Not only this, also to simultaneously maintain the strict Service Level Agreements (SLAs) for throughput, latency, and other metrics necessary to meet Quality of Service (QoS) standards.

In FaaS, resources are allocated dynamically in response to incoming requests, charging users only for the compute time used. These benefits stem from autoscaling, which refers to the dynamic adjustment of compute resources based on application demand from the user side. Autoscaling in cloud-native applications, specifically in Kubernetes-based deployments, is most commonly implemented through controllers. These controllers are usually Vertical Pod Autoscaler (VPA) and Horizontal Pod Autoscaler (HPA), monitoring the resource utilization metrics mainly including CPU and memory usage, while scaling the pods (number of application instances) accordingly [1], [3]. Autoscaling helps cloud providers to maintain performance KPIs under fluctuating workloads, optimizing the infrastructure usage, and controlling operational costs. This reactive nature presents critical limitations in today's dynamic computing environments. Traditional autoscalers often lead to cold starts [4], latency spikes, and resource under-provisioning during sudden workload fluctuations, while causing over-provisioning and resource waste during demand overestimation [5]. With cloud workloads becoming increasingly bursty, non-stationary, and shaped by diverse end-user behavior and time-of-day effects [6], reactive approaches struggle with partial observability, delayed feedback, and multi-dimensional optimization objectives.

Recent Machine Learning (ML) and Reinforcement Learning (RL) approaches have attempted to address these challenges through predictive models including LSTM networks [7] and RL algorithms such as Q-Learning [8], Deep Q Networks (DQN), and Proximal Policy Optimization (PPO) [1]. However, these methods remain limited by model drift, forecast errors, inability to generalize to unseen patterns, and evaluation primarily on synthetic traces with limited metrics [9]. Many approaches decouple prediction from autoscaler operation, optimizing for forecast accuracy rather than operational performance.

To overcome these limitations, we propose a novel autoscaling framework that integrates prediction and control through a PPO agent operating in a multi-dimensional discrete action space. Our approach enables coordinated scaling decisions across multiple resource dimensions (replicas, CPU, memory) within a single policy framework while dynamically adjusting learning parameters during training. The system incorporates

978-3-903176-75-1 ©2025 IFIP

three key contributions: (1) a custom double-stacked LSTM architecture integrated within the PPO policy network to capture both short-term bursts and long-term traffic dependencies, (2) an attention mechanism that identifies critical scaling moments in recent history, and (3) an SLA-aware reward function that balances latency compliance, success ratio, and resource efficiency. Unlike prior works where forecasting operates separately, our framework tightly integrates workload prediction with the RL loop, enabling truly proactive resource management.

The rest of this paper is structured as follows: Section II reviews relevant literature and state-of-the-art approaches. Section III presents the theoretical foundations and system model. Section IV details the methodology, experimental setup, and results. Section V discusses implications and future directions.

II. Related Work

Over the past five years, RL approaches for edge computing autoscaling have evolved from basic value-based methods to sophisticated hybrid architectures, yet significant gaps remain in achieving SLA-aware proactive scaling. Lee et al. [10] applied DQN for server instance scaling in multi-access edge computing environments, demonstrating the potential of value-based RL for dynamic resource allocation. However, their approach was limited to VM monitoring metrics without considering network conditions and lacked validation with microservice architectures. Gan et al. [11] proposed a PPO-based Markov Decision Process (MDP) with discrete actions, achieving 86% improvement over Q-Learning, demonstrating the superiority of policy gradient methods, yet their single-dimensional action space prevented coordinated optimization across multiple resource parameters. Benedetti et al. [12] deployed a Q-Learning framework based on CPU usage for edge autoscaling without significantly affecting latency, however, their evaluation was constrained to limited iterations and synthetic workloads.

The shift toward hybrid ML-RL approaches began with Ma et al. [8] introducing fuzzy Q-Learning with LSTM using differential evolution for hyperparameter selection, achieving better performance in execution time and energy consumption while maintaining optimal CPU usage. Moreover, their fuzzy logic integration introduced significant computational overhead unsuitable for resource-constrained edge environments, and their reward function prioritized energy consumption over SLA compliance, limiting applicability to latency-sensitive applications. Panda and Sarangi [13] developed FaaSCtrl, an Advantage Actor Critic (A2C)-based comprehensive latency controller that tunes Linux scheduling parameters (process priorities and CPU affinities) to manage all latency components (mean, median, standard deviation, tail latency), achieving 36.9% improvement in tail latency and 44.6% in response latency standard deviation. While their comprehensive latency modeling was innovative, their system-level approach operated below the application layer, limiting direct applicability to containerized microservices. Femminella and Reali [6] implemented PPO-based autoscaling in Kubernetes with cyclic time-of-day encoding (sin/cos functions) integrated with HPA, memory, and CPU usage metrics, comparing DQN, A2C, and PPO algorithms, representing significant progress in temporal feature engineering, yet their action space remained limited to traditional HPA parameters without forecasting integration for proactive scaling. Most critically, Agarwal et al. [14] introduced DRe-SCale, a PPO model integrated with LSTM for function autoscaling in partially observable environments, achieving 13-18% improvement over threshold-based approaches. While their recurrent architecture addressed temporal dependencies and partial observability through POMDP formulation, their LSTM integration was shallow (single-layer) and lacked attention mechanisms for critical pattern recognition.

Despite these advances, existing approaches exhibit fundamental limitations: single-dimensional action spaces preventing coordinated multi-parameter optimization, shallow temporal modeling insufficient for capturing both short-term bursts and long-term patterns in edge workloads, decoupled forecasting that optimizes prediction accuracy rather than operational performance, and reward functions that inadequately integrate strict SLA latency constraints essential for edge computing applications. Our RL-based autoscaler addresses these limitations through a novel multi-dimensional discrete action space enabling coordinated control over HPA CPU targets, throughput multipliers, and policy enhancements, integrated with double-stacked LSTM architecture and learned attention mechanisms for deep temporal modeling, direct forecasting integration within the RL pipeline for truly proactive scaling, and SLA-aware reward formulation with explicit 15ms latency constraints, evaluated comprehensively using real Azure Functions production traces across diverse workload patterns.

III. System Model

The search for intelligent and robust autoscaling in cloud and edge computing has led to research on learning-based resource controllers. Traditional methods, as discussed in previous sections, can be improved in terms of better scalability and computational overhead. Policy-Gradient RL [15] and, in particular, PPO have emerged as a state-of-the-art approach for efficient control in partially observable and high-dimensional domains. Furthermore, a double-stacked LSTM model is integrated within the PPO policy, feature-engineered with cyclic encoding and a custom attention mechanism, to enhance forecast accuracy and temporal resolution. The system architecture for our proposed approach has been shown in Fig. 1.

A. Reinforcement Learning Configuration and Observation Space

For this configuration, the autoscaler is formulated as an RL agent interacting with a dynamic environment. At each decision step, the agent observes the current system state and selects an action represented as a multi-dimensional discrete vector. Each element of this vector corresponds to a distinct control policy or scaling parameter, as summarized in Table I

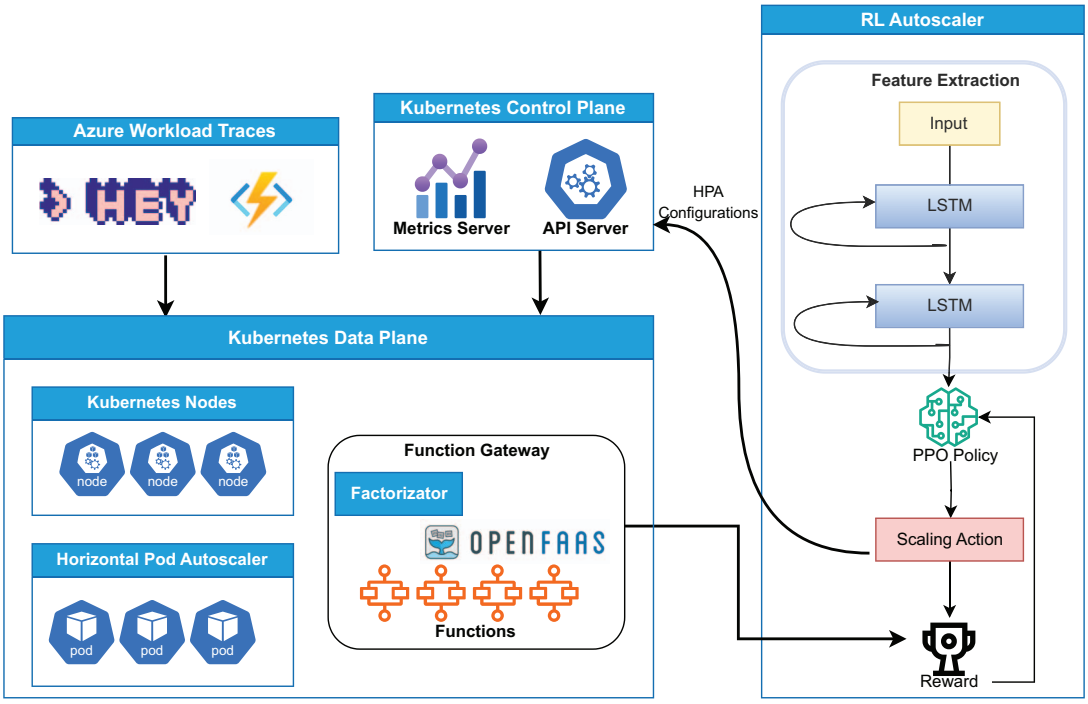

Fig. 1. System Architecture

and formally defined in Eq. (1). The observed state vector, denoted as $\mathbf{s}_t$ in Eq. (2), aggregates both present and historical system metrics. To be precise, it includes measured latency, replica count, per-pod RAM and CPU usage, total cluster resource consumption, current and forecasted request rates, HPA CPU target, observed success ratio, and cyclic encoding of the time-of-day.

$$\mathbf{a}_t = \left(a_t^{\text{cpu}},\ a_t^{\text{scale}},\ a_t^{\text{enh}},\ a_t^{\text{mode}}\right) \tag{1}$$

Where each element $a_t^{(\cdot)}$ is taken from a set of possible discrete actions for its respective control parameter.

$$\begin{aligned}\mathbf{s}_t = [\ &\text{latency}_t,\ \text{replicas}_t,\ \text{cpu}_t,\ \text{ram}_t,\ \text{requests}_t,\ \text{total\_cpu}_t,\\ &\text{total\_ram}_t,\ \text{success\_ratio}_t,\ \text{HPA}_t,\ \text{throughput\_mult}_t,\\ &\text{enhancement}_t,\ \cos(\theta_t),\ \sin(\theta_t),\ \text{forecast}_t]\end{aligned} \tag{2}$$

The selected state features ensure the RL agent receives a complete and actionable overview of the system. Latency and success ratio directly represent SLA fulfillment, while replicas, HPA target, and policy flags inform the agent about recent scaling actions. Including per-pod and total resource usage (CPU and RAM) provides visibility into both local and overall resource usage. Real-time and forecasted requests allow for both immediate and proactive responses. Moreover, cyclic time-of-day features help the agent recognize and adapt to regular workload patterns.

TABLE I
MULTI-DIMENSIONAL DISCRETE ACTION SPACE

Action	Parameter	Choices	Mapping
a_t^{cpu}	HPA CPU Target	0–4	(10, 30, 50, 70, 90)%
a_t^{lr}	LR Schedule	0–2	0: keep, 1: dec., 2: inc.
a_t^{tm}	Throughput Mult.	0–2	$\times 1$, $\times 2$, $\times 3$
a_t^{enh}	Enhancement	0–2	0:none, 1:moderate, 2:aggressive

B. Proximal Policy Optimization with Multi-Dimensional Discrete Actions

For addressing the multi-dimensional nature of autoscaling, simultaneous control of HPA targets, enhancement modes, and scaling multipliers, we extend classical PPO to operate over a discrete multi-action space. Through this, the agent can take coordinated actions across several control knobs within each epoch, without treating each scaling decision in isolation. Below, the Kullback-Leibler (KL) regularization (i.e., the core component of the model, penalizing the large deviations between the updated policy and the previous policy during RL iterations [16]), with entropy and surrogate objective, is given in Eq. (3). The entropy and KL penalties stabilize policy updates and encourage diverse exploration.

$$\begin{aligned}\mathcal{L}_{\text{PPO}}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\Big)\Big]\\ - \lambda_t \cdot \mathcal{H}\left[\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right] + \beta_t \cdot \text{KL}\left(\pi_\theta, \pi_{\theta_{\text{old}}}\right)\end{aligned} \tag{3}$$

Where,

$$r_t(\theta) = \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta_{\text{old}}}(\mathbf{a}_t|\mathbf{s}_t)}$$

$$\hat{A}_t = \text{Advantage estimate at time } t$$

$$\lambda_t, \beta_t = \text{entropy and KL regularization coefficients}$$

Standard PPO uses the clipping threshold ϵ to indirectly limit policy divergence; however, explicit KL regularization can be added as an extra penalty term. In this formulation for Eq. (3), both the clipping objective and a KL penalty are included for greater stability and to further drive policy updates. This is a valid PPO variant and does not replace the effect of the clipping threshold; rather, it complements it.

The learning rate is annealed using a Cosine schedule, given in Eq. (4). Cosine annealing is selected to enable the agent to learn fast initially and fine-tune later. This approach has demonstrated empirical superiority over step or exponential decay [17], partly because cosine annealing requires only desired initial and final learning rates, simplifying hyperparameter search [18]. Alternative scheduling algorithms were also considered; however, this method showed a balance between ease of use and training performance.

$$\eta(p) = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \cdot \frac{1 + \cos(\pi(1-p))}{2} \tag{4}$$

Where, p is the training progress and $\eta_{\min}, \eta_{\max}$ are the hyperparameters.

C. Reward Function Formulation

To balance SLA compliance ($L_{\text{SLA}} = 15$ ms), operational stability, and resource efficiency, we developed the composite reward function, as shown in Eq. (5) through systematic experimentation. This formulation emerged from multiple trial-and-error refinements where alternative approaches, such as linear penalties and unbounded terms, exhibited inferior convergence and stability. The final design provides balanced gradients for policy learning.

$$r_t = \gamma_1 \left[1 - \left(\frac{\text{latency}_t - L_{\text{SLA}}}{L_{\text{SLA}}}\right)^2\right] + \gamma_2 \,\text{success\_ratio}_t + \gamma_3 \,\exp\left(-\left(\frac{\text{cpu}_t - U^*}{U^*}\right)^2\right) + \gamma_4 \,\mathcal{R}_{\text{reg}}(\cdot) \tag{5}$$

The first term in Eq. (5) applies latency compliance using an inverted quadratic penalty by maximizing reward near L_{SLA} and sharply penalizing the deviations. The success ratio component $\gamma_2 \,\text{success\_ratio}_t$ rewards the request success rate, thereby maintaining throughput reliability. Resource efficiency (3rd term) is promoted by using a Gaussian form that peaks at target utilization U^*. The regularization component $\gamma_4 \,\mathcal{R}_{\text{reg}}(\cdot)$ integrates penalties for scaling oscillations, forecast deviations ($\hat{N}_t$ from section IV-B), and irregular resource usage. Finally, tunable weights γ_i balance these objectives across deployment scenarios, with values optimized during validation.

D. Sequential Feature Extraction

Autoscaling requires extracting meaningful insights in the form of features from time-dependent and dynamic input streams. In particular, edge computing workloads typically exhibit both short-term bursts (due to microservice traffic) and longer recurring patterns (e.g., scheduled events and daily usage patterns) [19]. Single-layer architectures and simple feed-forward networks can typically be insufficient for modeling these temporal dependencies because they do not possess memory of previous inputs [20], especially when autoscaling actions must predict both immediate changes and trends that extend for hours or even days. To highlight the most relevant timesteps in recent history, i.e., sudden load spikes or trend shifts, we implement a learned attention mechanism by assigning a weight to each hidden state in the input window of size T, enabling the agent to focus on critical moments. By using this technique, there is a direct interpretability advantage, i.e., the agent's scaling decision can be traced to the timesteps in history it attended the most.

$$e_t = \mathbf{w}_a^\top \mathbf{h}_t^{(2)} + b_a \tag{6}$$

$$\alpha_t = \frac{\exp(e_t)}{\sum_{k=1}^{T} \exp(e_k)} \tag{7}$$

$$\mathbf{c}_{\text{attn}} = \sum_{t=1}^{T} \alpha_t \, \mathbf{h}_t^{(2)} \tag{8}$$

In the attention mechanism, e_t is the attention score assigned to each hidden state $\mathbf{h}_t^{(2)}$ in the input window, computed as a learned linear combination (Eq. (6)). These scores are normalized using a softmax function to produce the attention weights α_t (Eq. (7)), which evaluate the importance of each timestep for the final context vector $\mathbf{c}_{\text{attn}}$ (Eq. (8)). To allow the agent to anticipate recurring periodic systems, e.g., scheduled events or regular load patterns (daily or hourly), the input at each time t is integrated with cyclic encoding, shown in Eq. (9) [6]. These engineered features are integrated with the main input, enabling the LSTM to differentiate among the same workloads occurring at different times of day.

$$\theta_t = 2\pi \cdot \frac{\text{step}_t}{\text{minutes\_per\_day}}$$
$$\mathbf{x}_t^{\text{cyclic}} = [\cos(\theta_t),\ \sin(\theta_t)] \tag{9}$$

Instead of treating temporal features and context separately, we propose a contextual fusion of double-stacked LSTM-attention output and cyclic encoding, shown in Eq. (10). This fusion strategy allows the agent to use high-level temporal awareness, local system state, and cyclic structure altogether for a better autoscaling performance. Apart from theory, we find that this integration helps the policy generalize better to unseen invocation patterns and position scaling actions with regular traffic cycles.

$$\mathbf{f}_t = \psi\left(\left[\mathbf{c}_{\text{attn}},\, \mathbf{x}_t^{\text{cyclic}},\, \mathbf{s}_t\right]\right) \tag{10}$$

Algorithm 1 Proactive SLA-Aware Autoscaling via PPO with Sequential Feature Fusion

Require: Environment $\mathcal{E}$, PPO agent π_θ, reward function $r(\cdot)$, forecast window T, episodes N_{ep}

Initialize policy parameters θ and value function ϕ
for episode $= 1$ to N_{ep} **do**
  Reset environment: $\mathbf{s}_0 \leftarrow \mathcal{E}.\text{reset}()$
  for $t = 1$ to $T_{\max}$ **do**
    Observe: System state $\mathbf{s}_t$ (latency, HPA, CPU, RAM, requests, etc.)
    Compute cyclic encoding: $\theta_t = 2\pi \frac{\text{step}_t}{\text{minutes\_per\_day}}$, $\mathbf{x}_t^{\text{cyclic}} = [\cos(\theta_t), \sin(\theta_t)]$
    Feature projection: $\mathbf{z}_t = \phi(\mathbf{W}_{\text{in}}\mathbf{x}_t + \mathbf{b}_{\text{in}})$
    Sequential modeling (double-stacked LSTM):
    $\mathbf{h}_t^{(1)}, \mathbf{c}_t^{(1)} \leftarrow \text{LSTM}^{(1)}(\mathbf{z}_t, \mathbf{h}_{t-1}^{(1)}, \mathbf{c}_{t-1}^{(1)})$
    $\mathbf{h}_t^{(2)}, \mathbf{c}_t^{(2)} \leftarrow \text{LSTM}^{(2)}(\mathbf{h}_t^{(1)}, \mathbf{h}_{t-1}^{(2)}, \mathbf{c}_{t-1}^{(2)})$
    Attention over window $\{\mathbf{h}_k^{(2)}\}$; Eq. (6), (7), (8)
    Feature fusion: $\mathbf{f}_t = \psi([\mathbf{c}_{\text{attn}}, \mathbf{x}_t^{\text{cyclic}}, \mathbf{s}_t])$
    Select action: $\mathbf{a}_t \sim \pi_\theta(\cdot|\mathbf{f}_t)$
    Apply action in environment $\mathcal{E}$
    Observe next state $\mathbf{s}_{t+1}$, compute reward r_t
    Store transition $(\mathbf{f}_t, \mathbf{a}_t, r_t, \mathbf{f}_{t+1})$
    if episode ends or $t = T_{\max}$ **then**
      break
    end if
  end for
  Policy Update: Use PPO (with entropy and KL regularization, cosine LR schedule) to update θ
end for
return Trained policy π_{θ^*}

Where $\psi(\cdot)$ is a learned feedforward mapping. This fused representation $\mathbf{f}_t$ is then used by the actor and critic heads to produce actions and value estimates.

Our RL-based autoscaler as a whole is differentiated by the integration of double-stacked LSTM layers and self-attention for better sequential modeling, explicit cyclic time encoding to capture recurring workload patterns, and the application of smoothing (only to the forecast feature) for robust and proactive scaling. The pseudo-code for RL-based autoscaler is shown in algorithm 1, and the parameters are reported in table II.

IV. Simulation Setup and Experimental Results

In this section, we discuss the simulation workflow, data preprocessing, workload generation, environment configuration, and results, keeping in view different metrics.

A. Data Preprocessing and Workload Segmentation

The workload consists of invocation traces collected from a production cloud system, i.e., Azure Functions [21]. Each trace entry contains real serverless function invocation logs, replicating true cloud workload bursts and variability. The data covers multiple days, out of which we have randomly used five days for training our model and two for its evaluation.

The most critical aspect of realistic load testing is the ability to inject workload patterns that accurately replicate real user traffic behavior, while also considering the closed-loop nature of the autoscaling in Kubernetes. For this reason, Hey load generator [22] is integrated into the system model. After calculating the smoothed expected requests $\bar{N}_t$ for the current timestamp t, the simulation issues a *hey* command to the target Kubernetes Service endpoint. Further, we deploy OpenFaaS [23], which acts as the gateway for function invocations, handling requests by replaying traces from Azure Functions. Among the deployed services is a custom microservice, factorizator, exposing an HTTP API for computationally intensive tasks, which in our case is the integer factorization.

TABLE II
Parameters and Hyperparameters Used in RL-based Autoscaler

Parameter	Value / Range
Batch size	128
Episode length	710
Number of training days	5
Number of test days	2
Observation window	1
Learning rate schedule	1×10^{-5} to 2×10^{-4}
Optimizer	Adam
Discount factor (γ)	0.995
GAE lambda	0.93
Entropy coefficient	0.01
PPO clipping ϵ	0.2
LSTM layers	2
LSTM hidden size	128
Attention mechanism	Yes
Forecast window size	3
SLA latency target (L_{SLA})	0.015 s
Desired CPU usage (U^*)	70%
Min/Max Replicas	1 / 200

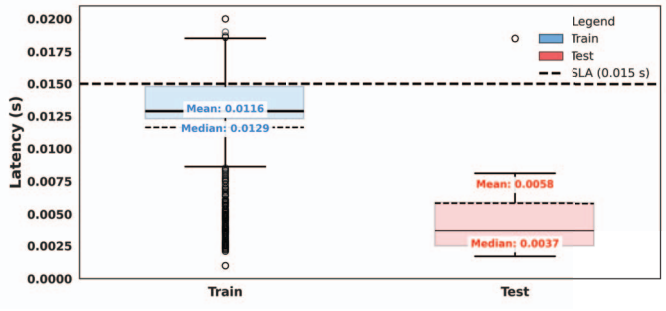

Fig. 2. Latency Distribution for Train and Test sets

B. Forecasting Integration

We integrate workload forecasting into state representation and reward design to enable proactive autoscaling. At each timestep t, a moving average forecast $\hat{N}_t$ of request rates is computed as shown in Eq. (11).

$$\hat{N}_t = \frac{1}{w} \sum_{k=1}^{w} N_{t-k} \tag{11}$$

Where N_{t-k} is the request count k steps before t, and w is the window size (refer to Table II). Concurrently, exponential smoothing is applied to raw counts N_t to mitigate noise while preserving trends, as shown in Eq. (12).

$$\bar{N}_t = \alpha N_t + (1 - \alpha)\bar{N}_{t-1} \tag{12}$$

The forecast $\hat{N}_t$ serves as both an input feature to the agent's state representation and an element of the reward function penalizing scaling decisions against the forecasted demand. The smoothed series $\bar{N}_t$ enhances signal stability for trend-sensitive operations.

C. Experimental Testbed

Simulations and RL-based autoscaling experiments were conducted on a local workstation equipped with an Intel Core i7-3770 CPU (8 threads, 3.4 GHz), 24 GB of RAM, and running Ubuntu Linux. Minikube was used to deploy Kubernetes locally (Client: v1.32.5, Server: v1.31.0), with all deployments and pods managed in a single-node setup. The RL training utilized Python 3.12, PyTorch 2.x, Stable Baselines3 2.3.0, and Gymnasium 0.29. All code was executed locally without the use of a GPU. The environment allows reproducible evaluation of our RL-based approach, as discussed in the sections below.

D. Latency Analysis

Figure 2 visualizes a comparative analysis of the latency distribution for both training and test phases, along with key statistical metrics and the target SLA threshold, i.e., 0.015 s. During the training phase, the latency values are spread with a mean of approximately 0.0116 s and a median of 0.0129 s, both well below the SLA boundary. The data points show some variability, especially at the 18-20 ms time window; however, the majority of samples consistently remain under the required threshold. In the test phase, performance further improves, with the mean and median latencies reduced to 0.0058 s and 0.0037 s, respectively. The test phase shows lower dispersed latency overall compared to the training phase; however, occasional latency spikes reach up to 18 ms. Notably, the model recovering its performance demonstrates resilience and continues to evaluate according to the learned policy.

E. Horizontal Pod Autoscaling CPU Utilization

Figure 3 visualizes the distribution of the HPA CPU target percentage across all steps during the training phase of our RL-based autoscaler. RL agent dynamically adapts the CPU target within the full action range (10% to 90%), frequently oscillating between intermediate and extreme values in response to changing invocation patterns. Frequent transitions between different CPU target levels indicate that the agent is neither stuck in a static single policy nor overfitting to any single workload segment. Rather, it uses both low and high CPU targets as needed:

- HPA CPU uses targets (e.g., 70–90%) during periods of increased demand to provision more resources, thereby preventing SLA violations.
- Lower targets (e.g., 10–30%) are favored when demand decreases, because the system reduces resource usage and improves efficiency.

F. Threshold Analysis

As shown in Figure 4, the success ratio for both training and testing phases consistently exceeds 97% across all scenarios. The success ratio is defined as the percentage of incoming requests that receive successful HTTP 200 responses out of the total requests sent in each simulation step. Specifically, our model achieves a training success ratio of 0.9943 and a testing success ratio of 0.9929.

G. Analysis of Reward and Normalized Forecast Error Against Number of Requests

Figures 5 and 6 provide an overall step-wise evaluation of the autoscaling agent during training and testing, respectively. Each figure contains three vertically aligned subplots: (a) reward r_t over time per step, (b) normalized forecast error per step, and (c) actual number of requests per step. During training, the reward displays high variance and considerable negative spikes, generally for the steps where the agent explored inefficiencies related to scaling decisions, either over-provisioning, under-provisioning, or exceeding the expected latency. As training advances, negative spikes become less frequent, and the reward signal stabilizes, signaling improved learning and adaptation. In the test phase, the reward trajectory is, however, much stable and predominantly positive, reflecting the agent's ability to generalize its learned policy to unseen demand patterns and maintain service efficiency. The mean reward values for the train and test sets are *1.632* and *2.212*, respectively, demonstrating post-training improvement. Further, to evaluate the forecasting accuracy, we use the normalized error at each timestep t, as shown in (13):

$$\mathrm{NE}_t = \frac{\hat{y}_t - y_t}{y_t} \tag{13}$$

Where $\hat{y}_t$ and y_t are the forecasted and actual request counts. This metric directly captures the direction and magnitude of prediction errors, with values near zero indicating high accuracy, positive values demonstrating overestimation, and negative values indicating underestimation. Most normalized errors are tightly grouped around zero (refer to Figures 5(b) and 6(b)), confirming that the LSTM-based predictor is largely unbiased for the test set and that the learned model generalizes well. In comparison, the training set exhibits more fluctuations, which coincide with periods of sudden changes in workload. These outliers, while expected in dynamic and real traces, have a limited impact due to the robust design of the autoscaling policy. The slightly higher under- and over-estimation in the training set is primarily due to the random allocation of training days, which, as shown in Figure 5(c), includes sudden bursts in incoming requests. These events challenge the forecasting model's accuracy, however the system nonetheless demonstrates rapid recovery and overall stability. The KDE plots in Figures 7 and 8 further illustrate the concentration of errors near zero, especially for the testing phase. Table III summarizes key quantitative metrics, such as MAE, RMSE, R^2, SLA compliance, and success ratio for both train and test phases. It is important to distinguish between success ratio and

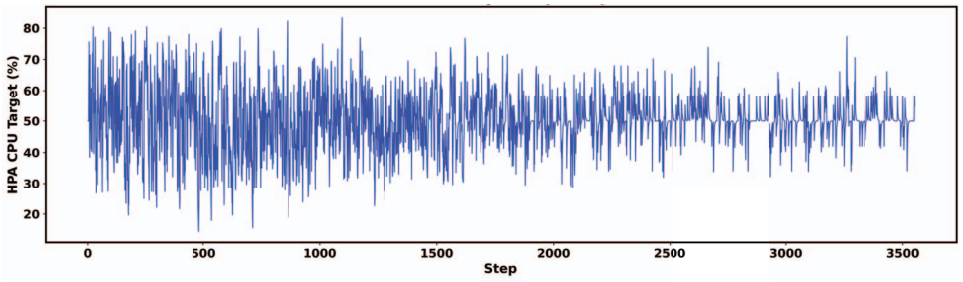

Fig. 3. HPA CPU Target Percentage for Train Set

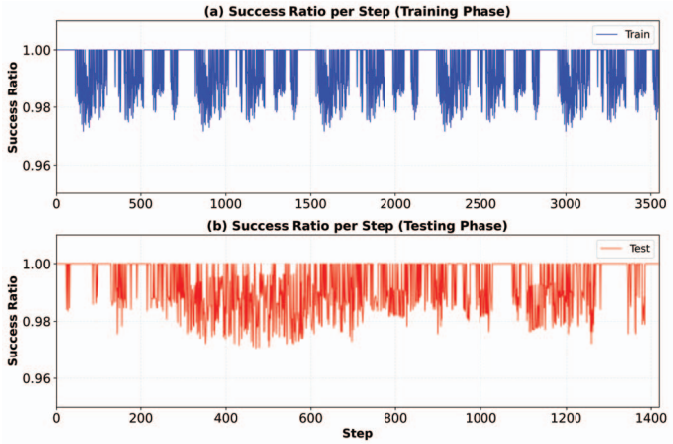

Fig. 4. Success ratio per step during (a) training and (b) testing phases. Average success ratios are 0.9943 (training) and 0.9929 (testing).

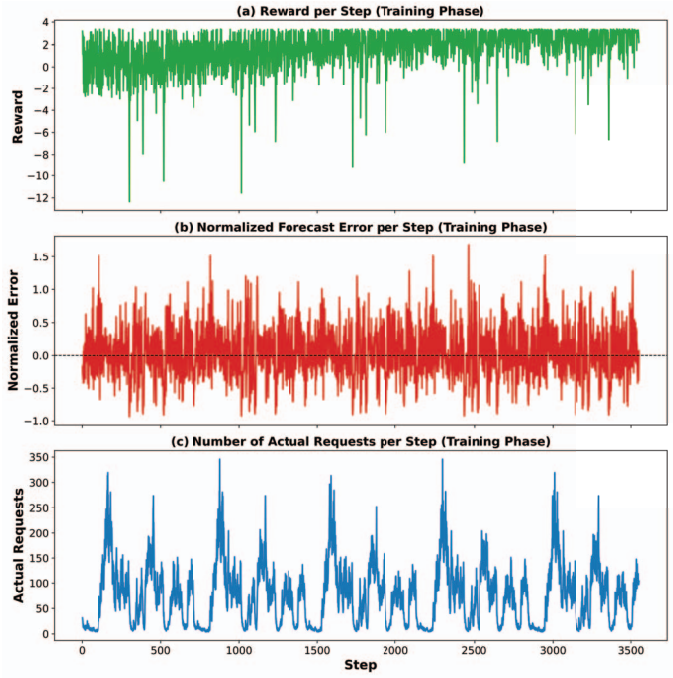

Fig. 5. Training phase analysis: (a) reward per step, (b) normalized forecast error, and (c) actual requests

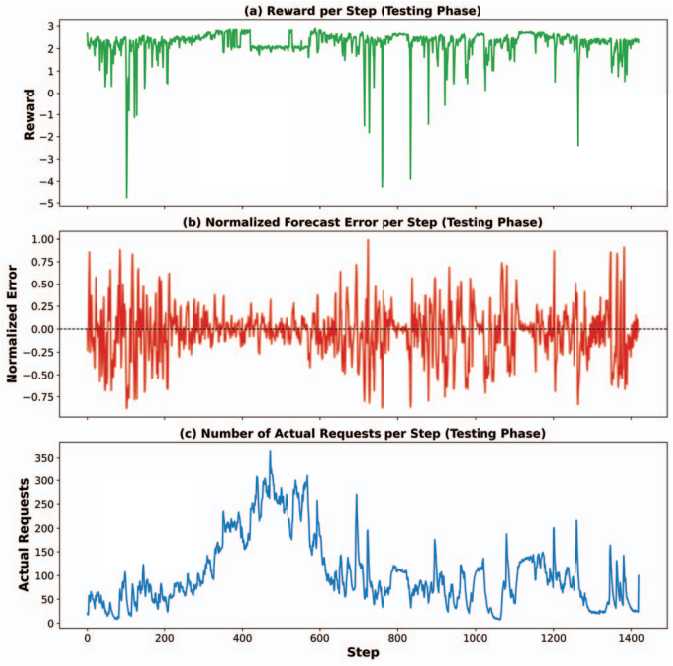

Fig. 6. Testing phase analysis: (a) reward per step, (b) normalized forecast error, and (c) actual requests

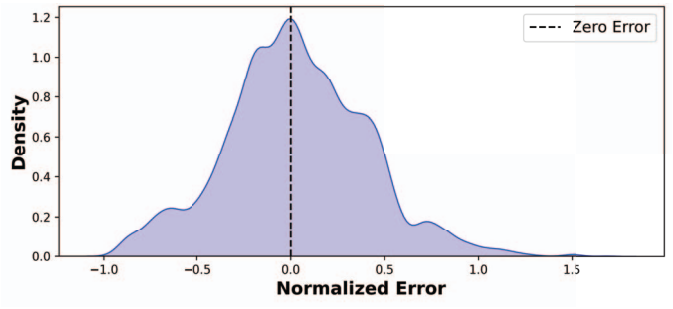

Fig. 7. Kernel density estimate (KDE) of the normalized error NE_t over the train set

SLA compliance metrics. The success ratio (>99%) measures the percentage of requests that receive successful HTTP 200 responses, indicating high system reliability. However, SLA compliance (80-87%) measures the percentage of requests that complete within our strict 15ms latency target.

TABLE III
COMPREHENSIVE PERFORMANCE METRICS FOR RL-BASED AUTOSCALER

Metric	Phase	MAE	RMSE	Other / Notable
Latency vs SLA	Train	0.0041	0.0062	SLA Compliance: 80.90%
	Test	0.0115	0.0118	SLA Compliance: 87.32%
Requests vs Forecast	Train	18.60	26.61	R^2: 0.7902
	Test	16.79	25.65	R^2: 0.8744
Throughput vs Forecast	Train	18.67	26.79	–
	Test	16.97	25.92	–
Success Ratio	Train	Mean: 0.9943	Std: 0.0078	$\geq$0.99: 68.51%
	Test	Mean: 0.9929	Std: 0.0084	$\geq$0.99: 62.54%
Reward	Train	Mean: 1.633	Std: 1.584	Min: -12.348, Max: 3.378
	Test	Mean: 2.213	Std: 0.641	Min: -4.759, Max: 2.898

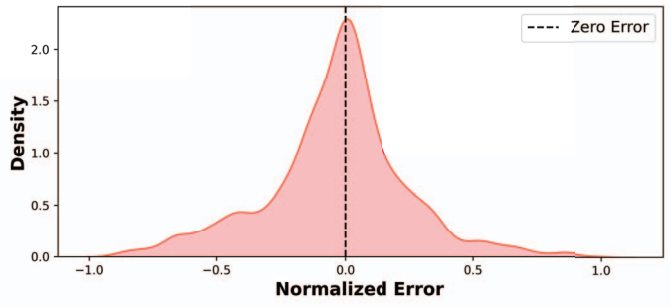

Fig. 8. Kernel density estimate (KDE) of the normalized error NE_t over the test set

TABLE IV
PERFORMANCE COMPARISON AGAINST STANDARD HPA (50%) AND BASELINE [6]

Metric	HPA (50%)	Baseline	Our Autoscaler
Max Latency (s)	0.912	0.707	0.018
Min Latency (s)	0.0020	0.0018	0.0017
Average Latency (s)	0.0722	0.0463	0.0058
Median Latency (s)	0.0692	0.0510	0.0037
Minimum Success Rate	0.862	1.000	0.975
Average Success Ratio	0.969	1.000	0.992
SLA Compliance (%)	0	0	87.32

H. Comparison With the Baseline

Table IV compares our RL-based autoscaler with standard HPA (50% CPU target) and the baseline [6] in terms of application latency, success ratio, and SLA compliance. Our autoscaler significantly reduces latency, lowering the average from 72.2 ms (HPA) and 46.3 ms (baseline) to 5.8 ms, and the maximum from 912 ms and 707 ms to 18 ms. Moreover, our autoscaler achieves 87.32% SLA compliance under the 15 ms threshold, whereas both other approaches remain at 0%. The autoscaler also maintains a 99.2% average success ratio, above HPA (96.9%) and slightly below the baseline (100%), with a 97.5% minimum success rate compared to 86.2% for HPA and 100% for the baseline. The improvement comes from integrating workload forecasting with stacked LSTM and attention mechanisms, allowing proactive scaling. In contrast, HPA reacts only to current CPU usage, and the baseline applies fixed thresholds without prediction, leading to higher latency and less efficient resource use.

V. DISCUSSION

The growing trend of cloud and edge computing applications has motivated researchers to address different challenges, including energy consumption, privacy, and efficient resource management. Within the domain of resource management, researchers and developers have focused significant effort on enhancing autoscaling techniques, specifically HPA, i.e., to update a workload resource automatically. To address these demands, we developed a PPO-based RL algorithm integrated with a double-stacked LSTM architecture and an attention mechanism, capable of forecasting incoming workload patterns, thereby mitigating the cold start and lagging adjustment issues. Our experimental methodology on the Azure invocation traces indicates the approach works as intended and maintains a balance between resource usage and performance, even under the bursty workload patterns without compromising the SLA agreements. Compared to threshold-based or single-policy solutions, our multi-dimensional discrete action space allows the agent to optimize multiple scaling parameters simultaneously, resulting in smoother transitions and a reduction in latency outliers. Further, the ability of policy to align the HPA CPU targets with observed and predicted workload patterns reflects a key advancement in bridging predictive analytics and adaptive control. Moreover, the random assignment of training and testing days contributes to exposing the autoscaler to diverse workload variations, crucial for evaluating generalizability in real-world deployments. The observed balance between rapid responsiveness and efficient resource utilization illustrates the efficiency of our RL-based autoscaler.

Despite the robustness of our RL-based autoscaler, several limitations remain. First, the current evaluation, conducted on a local single-node Kubernetes cluster, does not capture the full spectrum of complexities present in distributed, multi-cluster production environments. Notably, the rare however, significant forecast errors predominantly occur during periods of sudden and extreme workload changes. Such rapid rises or drops in demand present a basic challenge for all time series models, including our double-stacked LSTM-based forecaster, as these patterns in question frequently lie outside the dis-

tribution during training. Although it is inherently complex to predict such rare events due to their unpredictability and limited historical examples, our RL-based autoscaler, nonetheless, significantly reduces their operational impact by rapidly adjusting resource allocation in subsequent steps, hence maintaining SLA compliance and system stability.

As future work, we aim to refine reward shaping, enhance exploration strategies, and improve policy interpretability for better adaptation to changing workloads. A critical next step involves extending our evaluation from the current single-node testbed to distributed, multi-cluster edge-cloud infrastructures. While our proof-of-concept demonstrates effectiveness in a controlled environment, comprehensive validation requires deployment across multi-region and multi-tenant configurations to capture the complexities of production systems, including network latency and distributed coordination challenges.

Conclusion

This paper presents a novel RL-based autoscaling framework combining PPO, LSTM, and attention-driven forecasting for addressing the demands of today's cloud-native and edge computing environments. Through experimental simulation on real invocation traces and extensive evaluation in a Kubernetes testbed, our RL-based autoscaler demonstrates SLA maintenance and efficient resource management. Moreover, concerning the operational reliability, the system consistently achieved an average success ratio exceeding 99% in processing incoming requests. The flexibility of designing and enabling coordinated control over multiple scaling parameters differentiates it from much of the prior work using heuristic and static thresholds. The integration of forecasting directly into the RL pipeline proves effective in predicting sudden changes in workload and mitigating latency spikes. Nonetheless, the work also underscores the necessity of further investigation into policy generalization, large-scale deployment, and explainable decision-making to fully realize the vision of autonomous, self-optimizing infrastructures.

Acknowledgment

This work has been supported by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY, and the MUR Extended Partnerships grant PE00000001 - RESTART.

Data Availability Statement

Results of our experiments as CSV files can be available on demand by emailing the corresponding author. While the input data from Azure traces is publicly available.

References

[1] M. Femminella and G. Reali, "Comparison of reinforcement learning algorithms for edge computing applications deployed by serverless technologies," *Algorithms*, vol. 17, no. 8, 2024.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," *Research advances in cloud computing*, pp. 1–20, 2017.

[3] H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart hpa: A resource-efficient horizontal pod auto-scaler for microservice architectures," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 2024, pp. 46–57.

[4] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–36, 2024.

[5] Z. Zhang, T. Wang, A. Li, and W. Zhang, "Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2022, pp. 866–871.

[6] M. Femminella and G. Reali, "Application of proximal policy optimization for resource orchestration in serverless edge computing," *Computers*, vol. 13, no. 9, 2024.

[7] M. Golec, S. S. Gill, H. Wu, T. C. Can, M. Golec, O. Cetinkaya, F. Cuadrado, A. K. Parlikad, and S. Uhlig, "Master: Machine learning-based cold start latency prediction framework in serverless edge computing environments for industry 4.0," *IEEE Journal of Selected Areas in Sensors*, 2024.

[8] X. Ma, K. Zong, and A. Rezaeipanah, "Auto-scaling and computation offloading in edge/cloud computing: a fuzzy q-learning-based approach," *Wireless Networks*, vol. 30, no. 2, pp. 637–648, 2024.

[9] Y. Gari, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, "Reinforcement learning-based autoscaling of workflows in the cloud: A survey," *arXiv*, 2020.

[10] D.-Y. Lee, S.-Y. Jeong, K.-C. Ko, J.-H. Yoo, and J. W.-K. Hong, "Deep q-network-based auto scaling for service in a multi-access edge computing environment," *International Journal of Network Management*, vol. 31, no. 6, p. e2176, 2021.

[11] Z. Gan, R. Lin, and H. Zou, "Adaptive auto-scaling in mobile edge computing: A deep reinforcement learning approach," in *2022 2nd International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. IEEE, 2022, pp. 586–591.

[12] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications," in *2022 IEEE international conference on pervasive computing and communications workshops and other affiliated events (PerCom Workshops)*. IEEE, 2022, pp. 674–679.

[13] A. Panda and S. R. Sarangi, "Faasctrl: A comprehensive-latency controller for serverless platforms," *IEEE Transactions on Cloud Computing*, 2024.

[14] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions," *IEEE Transactions on Services Computing*, 2024.

[15] P. S. Thomas and E. Brunskill, "Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines," *arXiv preprint arXiv:1706.06643*, 2017.

[16] N. Vieillard, T. Kozuno, B. Scherrer, O. Pietquin, R. Munos, and M. Geist, "Leverage the average: an analysis of kl regularization in reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 12 163–12 174, 2020.

[17] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," 2017. [Online]. Available: https://arxiv.org/abs/1608.03983

[18] Z. Liu, "Super convergence cosine annealing with warm-up learning rate," in *CAIBDA 2022; 2nd International Conference on Artificial Intelligence, Big Data and Algorithms*. VDE, 2022, pp. 1–7.

[19] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey," *Computer Communications*, vol. 209, pp. 120–150, 2023.

[20] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[21] Microsoft Azure, "Azure functions invocation trace 2021," https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md, 2021, accessed: 2025-06-05.

[22] J. Rakyll, "hey: Http load generator, apachebench (ab) replacement," https://github.com/rakyll/hey, 2016, accessed: 2025-06-05.

[23] OpenFaaS, "Openfaas - serverless functions made simple," https://www.openfaas.com/, accessed: 2024-06-12.