STAR: Spatial-Temporal Autoscaling for Cloud Applications with Deep Reinforcement Learning

Zhengxin Fang[a], Hui Ma[a], Gang Chen[a], Shiping Chen[b]

[a]*School of Engineering and Computer Science & Centre for Data Science and Artificial Intelligence, Victoria University of Wellington, Wellington, New Zealand*
[b]*CSIRO Data61, Sydney, New South Wales, Australia*

Abstract

Autoscaling is an important technique for cloud computing that dynamically adjusts resources allocated to cloud applications in response to fluctuating user requests to maintain Quality of Service (QoS) and adhere to a given budget. Recent advancements in Deep Reinforcement Learning (DRL) have shown promise in achieving effective autoscaling approaches. However, prior DRL-based approaches struggle to simultaneously consider the spatial dependencies within an application and the changing historical workload patterns, limiting their ability to make accurate scaling decisions. Moreover, existing approaches lack the fine-grained resource adjustment, leading to suboptimal autoscaling performance. To address these limitations, we propose a new DRL-based autoscaling approach with a novel spatial-temporal autoscaling policy, which jointly captures spatial and temporal features of cloud applications by Graph Neural Networks and Transformers. Meanwhile, this policy enables fine-grained resource adjustment. Extensive experiments on real-world user request traces show that the proposed approach significantly outperforms existing state-of-the-art methods, achieving up to a 78.23% reduction in mean response time without violating the cost budget.

Keywords: Deep reinforcement learning, Autoscaling, Graph neural network, Transformer, Cloud Computing

Email addresses: zhengxin.fang@ecs.vuw.ac.nz (Zhengxin Fang), hui.ma@ecs.vuw.ac.nz (Hui Ma), gang.chen@ecs.vuw.ac.nz (Gang Chen), shiping.chen@data61.csiro.au (Shiping Chen)

1. Introduction

Cloud computing enables on-demand resource allocation through Virtual Machines (VMs) (Wang et al., 2020). Modern cloud applications increasingly adopt microservice architectures, decomposing complex systems into loosely coupled services instantiated by containers (Blinowski et al., 2022; Li et al., 2023). Fig. 1 (a) illustrates an example cloud application composed of four services, where the "front-end" handles user requests, "cart" and "browse" provide independent business functions, and "check" performs final processing. Each service is implemented by one or more containers, as shown in Fig. 1 (b).

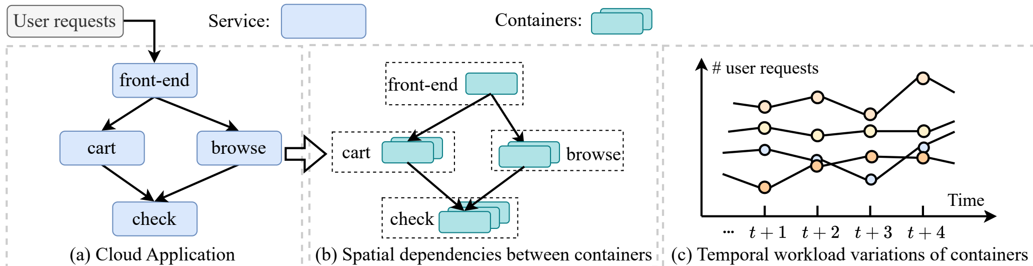

Fig. 1: The spatial and temporal features of an example cloud application

In practice, resources allocated to containers must be dynamically adjusted in response to workload fluctuations to maintain Quality of Service (QoS) while discouraging the VM rental cost are over the budget (Chen et al., 2024). However, spatial dependencies among containers (Fig. 1 (b)) and temporal workload variations (Fig. 1 (c)) significantly increase the complexity of resource allocation (Meng et al., 2022; Garí et al., 2024; Fang et al., 2023a; Shi et al., 2023; Nouri et al., 2019). To this end, autoscaling policies are widely adopted to dynamically adjust container resources through *scale-up* (resource expansion) and *scale-down* (resource reduction) operations.

Prior studies have explored either manual or automated design of autoscaling policies. Manually designed autoscaling policies (Amazon, 2022; Cheng et al., 2023; Burns et al., 2019) rely on expert knowledge to define fixed autoscaling rules. These manually designed autoscaling policies lack adaptability to a dynamically changing cloud environment. *Deep Reinforcement Learning* (DRL) is a promising approach for automatically designing and improving autoscaling policies (Shi et al., 2023; Luo et al., 2024; Meng et al., 2022, 2023), where sophisticated autoscaling policies are conveniently

modeled by Deep Neural Networks (DNNs) and effectively trained by reinforcement learning algorithms. These DRL-based approaches can learn useful autoscaling policies with strong adaptability and generalization performance in dynamic cloud environments.

Despite the advantages of DRL-based approaches, several issues remain unaddressed in existing work (Shi et al., 2023; Bai et al., 2024; Meng et al., 2022, 2023). (1) Existing DNN-based autoscaling policies cannot explicitly capture both the spatial dependencies within an application and historical workload information inherent in each container when making autoscaling decisions. Leveraging information on spatial dependencies can facilitate more accurate identification of containers that require scaling (Meng et al., 2022), while historical information of each container aids in determining the appropriate autoscaling resources (Luo et al., 2022b,a). (2) In existing DNN-based autoscaling policies, each output neuron of the policy network typically corresponds to a separate service. Due to this reason, these policies can only make autoscaling decisions for services instead of finer-grained containers. While service-level autoscaling adjusts all containers uniformly, container-level autoscaling provides fine-grained control over individual containers. Some practical systems (e.g., Kubernetes[1]) allow each service to be instantiated with heterogeneous containers (Fang et al., 2026; Baresi et al., 2021), thus container-level autoscaling is critical to accurately adjust the resources of containers within a service to reducing resource wastage (Bao et al., 2019; Wen et al., 2024).

In this paper, we propose Spatial-Temporal Autoscaling approach based on DRL, named *STAR*. A novel spatial-temporal autoscaling policy is implemented in *STAR*. To address the first issue, we design a spatial feature encoder and a temporal feature encoder in the policy neural network. Specifically, a Graph ATtention network (GAT)-based spatial feature encoder is designed to capture the complex spatial dependencies among containers within an application. Moreover, a Transformer-based (Vaswani et al., 2017) temporal feature encoder is introduced to capture the historical workload pattern of each container. Neural networks have been explored in other engineering domains for time-series data analysis (Sabry, 2021; El-Zathry et al., 2025; Wen et al., 2019; Wan et al., 2023). The self-attention mechanism allows Transformers to model long-term temporal dependencies more effectively than tra-

[1]https://kubernetes.io/

ditional neural networks (SABRY et al., 2017; Sabry et al., 2020).

Based on the learned container embeddings, we design a *hierarchical action network* to generate autoscaling actions. Formally, the hierarchical action network consists of a *Container Selector* and a *Scale Generator*. The Container Selector generates a high-level decision to select a target container for autoscaling, while the Scale Generator makes a low-level decision to determine the amount of resource to adjust for the target container. This hierarchical deep model allows the autoscaling policy to adapt to the changing number of containers, and make autoscaling decisions for specific containers (containers-level autoscaling), delivering an accurate and scalable solution for autoscaling. In this paper, we adopt an efficient and effective DRL algorithm, i.e., Evolutionary Strategy-based Reinforcement Learning (ESRL) (Salimans et al., 2017), to train the autoscaling policy.

In summary, this paper makes the following contributions:

- We propose a spatial feature encoder and a temporal feature encoder to capture the bi-directional spatial dependencies among containers and their historical workload information. These encoders allow us to learn informative embeddings that are vital for inducing effective autoscaling decisions.
- We propose a hierarchical action network to generate autoscaling actions, which can adapt to the changing number of containers and make fine-grained (container-level) autoscaling decisions.
- We conduct extensive experiments based on real-world traces of user requests to examine the performance of *STAR*. Our experimental results show that *STAR* outperforms four state-of-the-art autoscaling approaches by enhancing the QoS while saving cost.

The rest of this paper is organized as follows. Section 2 reviews related work on autoscaling approaches. Section 3 formulates the problem of autoscaling cloud applications. Section 4 presents the details of our proposed *STAR* approach. Section 5 reports extensive experiments evaluating the performance of *STAR*. Section 6 provides further analysis of the proposed approach. Finally, Section 7 concludes the paper and outlines directions for future work.

2. Related Work

In this section, we review recent autoscaling approaches. Specifically, the review starts with a summary of manually design (heuristic-based) autoscaling policies, followed by DRL-based approaches for designing autoscaling policies automatically.

2.1. Heuristic-based Autoscaling

Herusitic-based autoscaling approaches such as AWS-Scale (Amazon, 2022) and HPA (Burns et al., 2019) rely on manually designed rules for making autoscaling decisions. Specifically, more containers are created if the resource utilization is higher than the upper threshold. On the other hand, some containers are terminated whenever utilization falls under the lower threshold.

Some heuristic-based approaches incorporate predictive approaches to enhance autoscaling decisions. For instance, PBScaler (Xie et al., 2024) is proposed to detect bottleneck services in an application. Subsequently, a genetic algorithm is applied to decide the number of containers required by the bottleneck service. ProScale (Cheng et al., 2023) is an autoscaling approach that leverages the effective and efficient Simple Moving Average (SMA) approach to predict future request workloads. Then, the resources of services are adjusted based on a greedy strategy according to the predicted workloads.

Graph Neural Networks (GNNs) are also adopted for workload prediction (Meng et al., 2023; Park et al., 2024). These approaches utilize GNNs and supervised learning to effectively predict the future workload of applications, then make autoscaling decisions based on manually designed rules and predicted workload.

The heuristic-based autoscaling approaches mentioned above require huge effort to design effective rules (Xing et al., 2022; Wang et al., 2020). Moreover, these heuristic-based approaches often struggle to generalize well across changing workloads.

2.2. Reinforcement Learning-based Autoscaling

Recent research (Shi et al., 2023; Bai et al., 2024; Meng et al., 2022; Garí et al., 2022, 2024; Zhang et al., 2020) showed that Reinforcement Learning (RL)-based autoscaling approaches can effectively learn *scaling policies* with strong adaptability to changing workloads.

For instance, an RL-based approach, named A-SARSA, is proposed (Zhang et al., 2020) to combine DNN-based workload prediction and the SARSA algorithm (Watkins and Dayan, 1992) to make autoscaling decisions based on

predicted workload. However, A-SARSA did not explicitly consider the complex dependencies between services within a cloud application, hindering the effectiveness of autoscaling (Li et al., 2023). Q learning-based autoscaling approaches (Garí et al., 2022, 2024) are proposed for workflow autoscaling, which considers the application structure while making autoscaling actions.

The RL-based approaches mentioned above often struggle to handle high-dimensional state spaces due to their limited feature extraction capabilities (Shi et al., 2023). To overcome this limitation, DRL-based autoscaling approaches have become popular in recent years by leveraging DNN and RL algorithms to learn embeddings and make effective autoscaling decisions. For example, a Deep Q-Network (DQN) (Mnih et al., 2015) based autoscaling approach, called HRA (Meng et al., 2022), is proposed to make holistic autoscaling decisions for applications, which is trained by Q-learning. Similarly, (Shi et al., 2023) proposes a DeepScale approach that integrates DQN and heuristic approaches to manage the scaling of application instances.

One advantage of RL-based approaches is their ability to train the autoscaling policy online. For example, Gari et al. (Garí et al., 2022, 2024) compare Q-learning and SARSA for online training in autoscaling. CoScale (Xu et al., 2022a) is an RL-based autoscaling approach that incorporates offline and online training together. A TD3-based DRL approach, named DRPC (Bai et al., 2024), is proposed to make autoscaling decisions based on the embedding of services learned by multiple distributed neural networks. Moreover, DRPC periodically retrained the policy during the online interaction with the environment.

The above RL-based autoscaling approaches either overlook the spatial characteristics of cloud applications or fail to capture the historical workload information of individual containers. Moreover, in existing studies, each output neuron of the policy network corresponds to a separate service, limiting the ability to make more fine-grained resource adjustments for each container, since the number of containers changes over time. To address these two issues, this paper aims to design a novel DRL-based autoscaling approach.

2.3. Summary

Motivated by prior limitations, the proposed STAR introduces two key methodological innovations that are expected to achieve superior autoscaling performance. First, *STAR* utilizes a novel Transformer-based temporal encoder to learn each container's historical workload trends and variations,

a design not explored in prior autoscaling research. This design is expected to enable accurate resource adjustments that reduce latency while avoiding unnecessary costs. Second, *STAR* enables fine-grained container-level autoscaling through a newly designed *hierarchical action network*, comprising a *Container Selector* and a *Scale Generator*. This design, never considered in prior approaches, allows *STAR* to support a varying collection of containers and make precise fine-grained scaling decisions that avoid over-allocation and reduce costs.

3. Problem Definition

In this section, we formalize the problem of autoscaling for cloud applications.

As shown in Fig. 2, each service of a cloud application is implemented by one or multiple containers. Let ins_i denote a container. In a cloud data center, containers are deployed in VMs, each of which can host multiple containers, while each VM is further deployed on a physical machine (PM) (Wang et al., 2020; Tan et al., 2020; Fang et al., 2023b,a). The deployment of containers and VMs utilizes the Best-Fit heuristic (Jangiti et al., 2020). This heuristic places each new container/VM on the VM/PM whose remaining resources most closely match its requirements, thereby minimizing resource fragmentation and reducing wastage (Wen et al., 2024; Srirama et al., 2020).

The optimization objective of this problem is twofold: (1) to enhance the QoS of applications and (2) to penalize when the application deployment costs exceed the allocated budget. This optimization objective differs from the existing literature (Bai et al., 2024; Shi et al., 2023; Garí et al., 2022, 2024), which mainly focus on reducing Service Level Agreement (SLA) (Patel et al., 2009) violations (Garí et al., 2022, 2024), improving resource utilization (Bai et al., 2024), or minimizing the overall cost (Shi et al., 2023). Our design reflects the real-world need (FinOps, 2025; Microsoft, 2024; Wang et al., 2020) to trade off application performance and cost.

The CPU resources allocated to containers primarily determine cloud applications' response time (Wang et al., 2020; Fang et al., 2023a), while the number of active VMs governs the overall rental cost. In practice, an application must support a large and dynamically changing number of user requests. As a result, an autoscaling policy is required to select target containers for autoscaling and determine the amount of scaling resources based on the monitored status of the cloud data center, see Fig. 2.

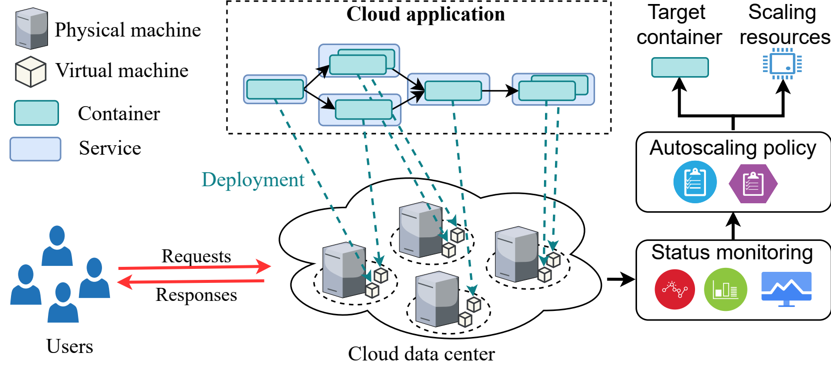

Fig. 2: The overview of application autoscaling in cloud data center

We evaluate the QoS of cloud applications by the mean response time over a time period (Shi et al., 2023; Cheng et al., 2023). Let $req_k = \{task_s, \ldots, task_e\}$ denote the k-th user request, where $\{task_0, task_1, \ldots, task_s\}$ represents a set of tasks with precedence relation. Each task is associated with a specific service, and is executed by one of the container instances belonging to that service. For example, $task_s$ is executed by a container of service s. Meanwhile, each container maintains a pending queue for incoming tasks to be processed, following existing studies (Shi et al., 2020, 2023; Yang et al., 2022; Wang et al., 2020). Although executing a single task in a container may lead to lower resource utilization, it is important to enhance system stability and isolated security for better application management (Shi et al., 2023; Fang et al., 2026). Incoming tasks are enqueued in a container upon arrival, and the first pending task in the queue is dispatched for processing once the container becomes available. In practice, lightweight in-memory systems (e.g., Redis Lists[2,3]) can efficiently manage request queues, ensuring ordered processing.

As assumed in existing studies (Wang et al., 2020; Shi et al., 2020; Yang et al., 2022), a container can execute at most one task at any time. As a result, the starting time for $task_s$ in request k on container ins_i is determined

[2]https://medium.com/@rahulpt11997/synchronising-micro-services-with-redis-a-kubernetes-case-study-af36298cbb91

[3]https://aws.amazon.com/blogs/database/build-with-redis-data-structures-for-microservices-using-amazon-memorydb-for-redis-and-amazon-ecs/?utm_source=chatgpt.com

by

$$st_s^i = \begin{cases} a_t + wt_k^i, & s = 0; \\ a_t + wt_k^i + ft_k^{pre_i}, & \text{otherwise.} \end{cases} \quad (1)$$

where a_t is the arrival time of req_k, wt_k^i is the waiting time of $task_s$ in the pending queue of container ins_i, while $ft_k^{pre_i}$ denotes the finish time of the predecessor container of ins_i in processing req_k.

Let ot_k^i denote the execution time of ins_i with one vCPU for a request req_k. We assume that a container starts execution only after all its preceding containers have completed and the required data have been fully transferred (Bai et al., 2024; Shi et al., 2023; Fang et al., 2026). Therefore, during the execution phase, containers do not incur additional communication or synchronization overhead and can be regarded as parallel. Although this assumption may overlook potential communication delays during execution, their practical impact is negligible compared with execution delay (Shi et al., 2023; Fang et al., 2026; Bai et al., 2024). Accordingly, the finished time of ins_i to process req_k is

$$ft_k^i = \frac{ot_k^i}{cpu_i} + st_k^i, \quad (2)$$

where cpu_i is the number of vCPU allocated to ins_i. As a result, the response time (rt_k) of req_k is given by:

$$rt_k = ft_k^e, \quad (3)$$

where ft_k^e is the finish time of the last container required in req_k.

Thus, the mean response time of an application over a time period T is

$$MRT(T) = \frac{\sum_{k \in REQ(T)} rt_k}{num(T)}, \quad (4)$$

where $REQ(T)$ is the set of user requests over a time period T, and $num(T)$ is the number of requests over a time period T.

The rental fee $cost_v$ of a VM instance VM_v is calculated by:

$$cost_v = price_v \times \frac{FT_{last}^v - ST_{first}^v}{3600}, \quad (5)$$

where FT_{last}^v and ST_{first}^v are the finish time of the last request and the start time of the first request executed in VM_v, respectively. The total cost $cost(T)$

of renting VMs over T is calculate by:

$$C(T) = \sum_{VM_v \in SET_{VM}} cost_v, \tag{6}$$

where SET_{VM} is a set of all VM instances. Thus, we define the violation degree of cost as

$$Vio(T) = \max\big(0, (C(T) - budget(T)\big), \tag{7}$$

where $budget(T)$ is the given budget over a time period T.

In summary, the optimization objective of autoscaling cloud applications can be formulated as

$$\min MRT(T) + \rho \cdot Vio(T) \tag{8}$$

where ρ is the penalty coefficient for exceeding the budget limit.

4. Proposed Approach: STAR

The overall process of using *STAR* for autoscaling is illustrated in Fig. 3, which is formulated as an RL problem, detailed in Section 4.1. In a cloud environment, the time-varying state is continuously monitored by the *STAR* agent. These fluctuations occur due to dynamically changing user requests, which in turn trigger resource adjustments or the creation and termination of containers. Based on the latest state, the spatial-temporal autoscaling policy of the *STAR* agent outputs autoscaling actions to accommodate changing workloads, as detailed in Section 4.2.

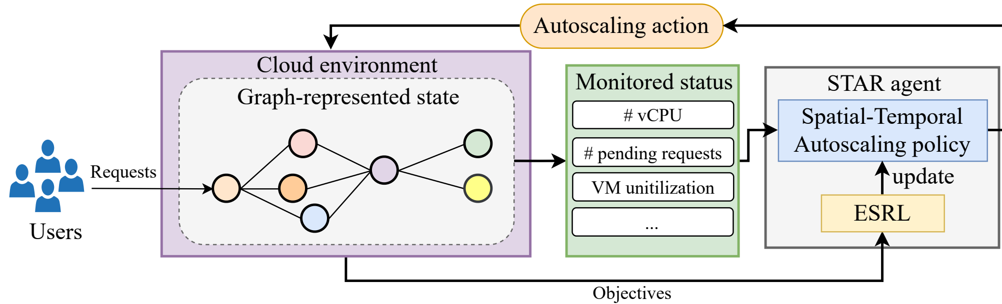

Fig. 3: The overall process of autoscaling by STAR

An RL algorithm, i.e., ESRL (Salimans et al., 2017), is utilized to train the spatial-temporal autoscaling policy of *STAR*, introduced in Section 4.3. This is because ESRL maintains a stable training process, requires few hyperparameters to tune, and is robust to hyperparameter variations (Huang et al., 2022; Salimans et al., 2017). Furthermore, recent studies have demonstrated its effectiveness in various cloud-related applications (Huang et al., 2022; Shen et al., 2024). Unlike other algorithms such as PPO, ESRL does not rely on step-wise rewards. Instead, it only uses the final return obtained at the end of an episode to guide the training process. Consequently, the return is defined as the negative of the optimization objective (Eq. 8), so that minimizing the objective corresponds to maximizing the return.

4.1. RL Formulation

We formulate the autoscaling process supported by STAR as an RL problem. As shown in Fig. 3, the cloud environment provides state s_t in the form of a graph at any decision time point t to the *STAR* agent. The *STAR* agent, in turn, performs autoscaling action a_t based on s_t, then the cloud environment transits to the next state s_{t+1}. The details of key components of this RL problem are provided as follows.

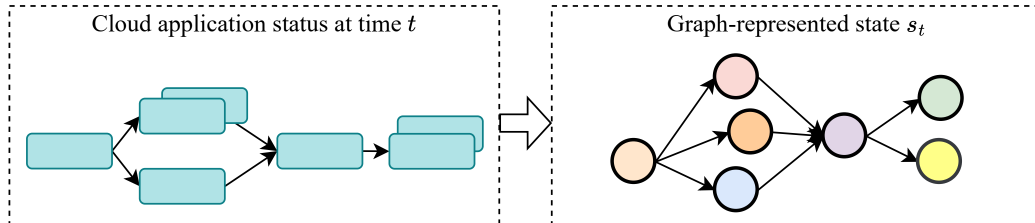

Fig. 4: Graph-represented state of an application

4.1.1. State

The state s_t denotes a snapshot of the cloud application at time t. As shown in Fig. 4, a cloud application status at time t is represented by a graph $\mathcal{G}_t = \langle \mathcal{V}_t, \mathcal{E}_t \rangle$. Specifically, a container ins_i is represented by a node $v_i \in \mathcal{V}_t$ in the graph, while edge $e_{i,j} \in \mathcal{E}_t$ represents the execution order between ins_i and ins_j. This graph-represented state evolves over time with incoming requests and autoscaling actions.

Let $\vec{h_i} = \{\Omega_i,\ \zeta_i,\ d_i,\ pending_i,\ art_i,\ vm_info_i,\ pm_info_i,\ predicted_i\}$ denote the feature vector of a container ins_i. The detailed description of

each feature in $vech_i$ is provided in Table 1. Following prior study (Bai et al., 2024), we adopt an existing prediction model (Xu et al., 2022b) to predict future workload based on current workload for the feature $predicted_i$. The workload prediction model is fixed after training with the historical workload traces in each scenario.

Table 1: Detailed description of the features.

Notation	**Description**
Ω_i	The allocated CPU and memory resources of container ins_i.
ζ_i	Available resource capacity of the VM instance that hosts container ins_i.
d_i	Number of neighboring containers connected to container ins_i in the graph.
$pending_i$	Number of requests currently waiting to be processed by container ins_i.
art_i	Average response time of container ins_i over the recent time window.
vm_info_i	Static and dynamic information of the VM instance that hosts container ins_i.
pm_info_i	Static and dynamic information of the PM instance that hosts the VM of container ins_i.
$predicted_i$	Forecasted workload intensity of container ins_i in the next scheduling interval.

Overall, the above features jointly characterize workload demand (e.g., predicted workload, pending requests, and response time), resource capacity at container/VM/PM levels, and inter-container dependencies. These features are consistent with those explored in prior studies (Fang et al., 2026; Shi et al., 2023), which have shown that they are sufficient for autoscaling decisions.

4.1.2. Action

An action a_t generated by the *STAR* agent at time t is defined as a tuple: $\langle target, scaling \rangle$. Here, *target* represents the container targeted for scaling. *scaling* indicates the specific scaling strategies, including horizontal scaling (create or delete containers), vertical scaling (increase or decrease the

vCPU of existing containers). This study focuses on the resource adjustment of vCPU in line with previous studies (Shi et al., 2023; Park et al., 2024), as vCPU is the dominant factor that impacts the response time of cloud applications (Park et al., 2024; Wen et al., 2025).

4.1.3. State Transition

The cloud environment transits from s_t to s_{t+1} after the *STAR* agent performs a_t. Fig. 5 shows an example of a state transition. In particular, the graph-represented state is transited from s_t to s_{t+1} when the *STAR* agent performs a_t to create a new replica of the target container by horizontal scaling.

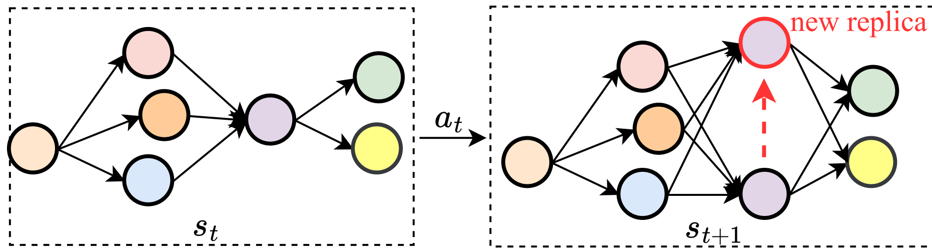

Fig. 5: An example of state transition

4.2. Spatial-Temporal Autoscaling Policy

Fig. 6 illustrates the overall structure of the proposed spatial-temporal autoscaling policy, consisting of three main components: (1) Spatial feature encoder, which captures the complex bi-directional spatial dependencies among containers; (2) Temporal feature encoder, which captures the historical workload information of a specific container; (3) Action Generation, which makes autoscaling decision by identifying the container *target* required for scaling and determining the specific scaling strategies (i.e., *scaling*) based on the embeddings learned by the spatial feature encoder and temporal feature encoder. Details of these three components are provided in the following.

4.2.1. Spatial Feature Encoder

The spatial feature encoder takes the graph-represented state as input and outputs the spatial embeddings of each container. To capture the spatial dependencies within an application, we stack *graph attention layers* (Velickovic et al., 2017) to construct a GAT network. Specifically, *graph attention*

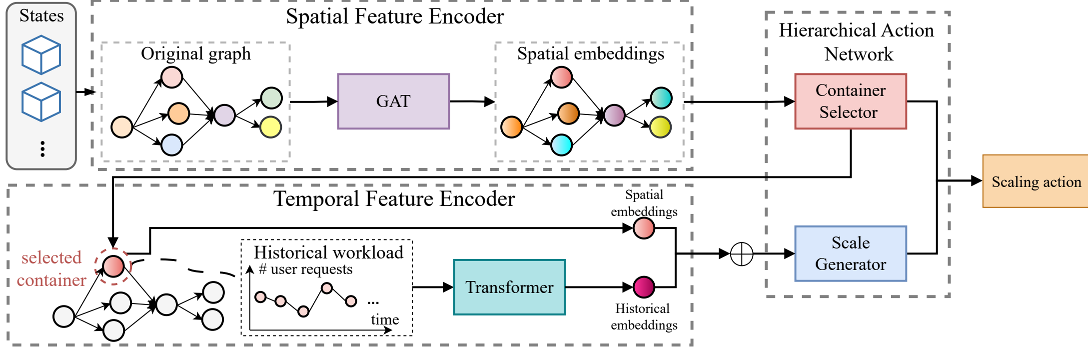

Fig. 6: The overall framework of spatial-temporal autoscaling policy

layer dynamically assigns attention weights to each node and its neighbors, allowing *STAR* to learn expressive spatial embeddings for each container by aggregating information from its neighbor containers with different weights.

Let $\mathbf{h} = \{\vec{h_1}, \vec{h_2} \ldots, \vec{h_N}\}$ denote the set of raw features for all the containers, where N is the number of containers, and $\vec{h_i}$ represents the raw features of container ins_i. Each *graph attention layer* first calculates the attention weight, then outputs a new feature vector, i.e., $\mathbf{h}' = \{\vec{h'_1}, \vec{h'_2} \ldots, \vec{h'_N}\}$. Specifically, the attention weight $\alpha_{i,j}$ indicating the weight between connection of ins_i and ins_j is calculated by

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}(\vec{a}^T[\mathbf{W}_l\vec{h_i} \| \mathbf{W}_l\vec{h_j}])\right)}{\sum_{o\in\mathcal{N}(i)} \exp\left(\text{LeakyReLU}(\vec{a}^T[\mathbf{W}_l\vec{h_i} \| \mathbf{W}_l\vec{h_o}])\right)}, \tag{9}$$

where $\mathbf{W}_l \in \mathbb{R}^{F' \times F}$ is a trainable weight matrix applied for linear transformation to every node. $\|$ indicates the concatenation operation. $\vec{a} \in \mathbb{R}^{2F'}$ is the trainable weight vector of a feedforward network. Following (Velickovic et al., 2017), LeakyReLU is applied for non-linear transformation. $\mathcal{N}(i)$ is the neighbor nodes of container ins_i.

We define the neighbor set of ins_i as $\mathcal{N}(i) = \{u \in \mathcal{V} \mid e_{u,i} \in E \vee e_{i,u} \in \mathcal{E}\}$. This implies that the information is aggregated not only from a container's succeeding nodes but also from its predecessor nodes, as shown in Fig. 7. This allows our proposed spatial feature encoder to capture complex spatial contexts within an application. The new features $\vec{h'_i}$ of ins_i is generated by a *graph attention layer* according to

$$\vec{h'_i} = \boldsymbol{\sigma}\Big(\sum_{j\in\mathcal{N}_i} \alpha_{i,j}\mathbf{W}_h\vec{h_i}\Big), \tag{10}$$

where $\boldsymbol{\sigma}$ is the sigmoid activation function that is applied for nonlinearity. $\mathbf{W}_h$ is a trainable linear transformation matrix. The GAT takes the whole graph as input and outputs spatial embeddings of each container after passing through multiple stacked *graph attention layers*.

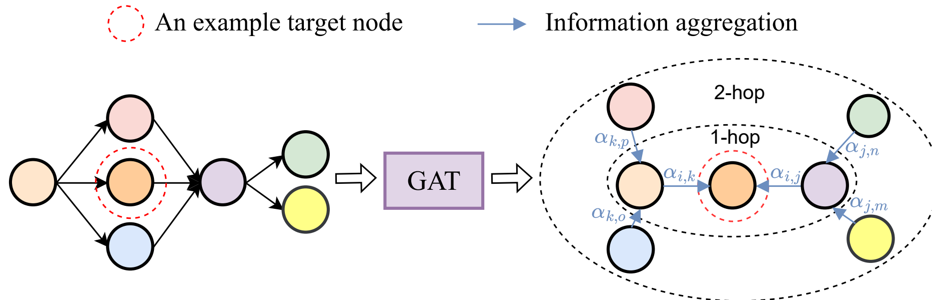

Fig. 7: Example of information aggregation by GAT with 1-hop and 2-hop neighbors.

4.2.2. Temporal Feature Encoder

The temporal feature encoder takes the historical workload of a container as input, and outputs the temporal embeddings of the corresponding container. In particular, each container records its historical workload information, that is the number of user requests it processes at all previous autoscaling decision points, as shown in Fig. 6. Then, a transformer structure is utilized to process the historical workload variations of a container.

Let $\vec{w} = \{w_1, w_2, \ldots, w_T\}$ denotes the historical workload of a container, where $w_t \in \mathbb{Z}$ indicates the number of user requests processed at time t. To effectively model the historical workload pattern of containers, $\vec{w}$ is taken as the input of the self-attention layer, and outputs the embeddings $\mathbf{E} = \{e_1, e_2, \ldots, e_t\}$ by

$$\mathbf{E} = \mathrm{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \tag{11}$$

where $\mathbf{Q} = \vec{w}\mathbf{W}_q$, $\mathbf{K} = \vec{w}\mathbf{W}_k$ and $\mathbf{V} = \vec{w}\mathbf{W}_v$. $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ are the trainable projection matrices for query, key, and value (Vaswani et al., 2017).

Finally, we apply mean pooling to $\mathbf{E} = \{e_1, e_2, \ldots, e_T\}$ to obtain the historical embedding $\mathbf{z} = \frac{1}{T}\sum_{t=1}^{T} e_t$. This design is motivated by the fact that autoscaling decisions rely on overall workload trends rather than a single time

step. Mean pooling provides a stable and unbiased summary of historical dynamics. The historical embeddings of a container capture the long-term pattern of its historical workload, guiding an accurate autoscaling decision to allocate reasonable resources to the container.

4.2.3. Hierarchical Action Network

To adapt to the changing number of containers and support *containers-level* autoscaling actions, we design a hierarchical action network to generate autoscaling actions, consisting of a Container Selector and a Scale Generator. It is noted that the hierarchical action network selects only one container per autoscaling decision to maintain the stability of the cloud environment and prevent abrupt resource fluctuations (Russo Russo et al., 2023; Anselmi et al., 2025; Mahmoudi and Khazaei, 2022). The detailed framework of the hierarchical action network is shown in Fig. 8.

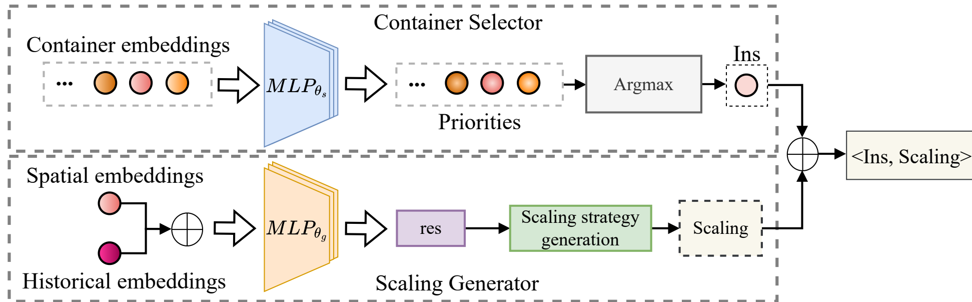

Fig. 8: The framework of action generator

Specifically, in Container Selector, a Multi-Layer Perceptron MLP_{θ_s} with trainable parameters θ_s takes the embedding of each container as input and outputs a *priority* of the corresponding container. A higher *priority* indicates the container requires scaling more urgently. As a result, the container with the highest *priority* (i.e., *target*) is selected for autoscaling. This design supports an arbitrary number of containers, enabling the identification of the container that most needs scaling.

After selecting a container $target$ for autoscaling, its spatial embeddings are concatenated with its historical embeddings, which are fed into another MLP (i.e., MLP_{θ_g}) with trainable parameters θ_g to determine $res \in [-m, +m] \cap \mathbb{Z}^+$. The res indicates the requested change of allocated resources for $target$ after scaling, and m is set as 4 in this paper based on

existing studies (Bai et al., 2024; Xu et al., 2022a). The value of m affects the trade-off between responsiveness and system stability, and setting $m = 4$ can handle many real-world workloads (van Baaren, 2009; Luo et al., 2022b; Laboratory, 2022; Cheng et al., 2023; Xie et al., 2024) while maintaining system stability by preventing drastic changes in resource allocation during autoscaling. This stability helps alleviate transient effects, as appropriately resource adjustments allow the system to adapt smoothly without sudden spikes or fluctuations (Russo Russo et al., 2023; Anselmi et al., 2025; Mahmoudi and Khazaei, 2022)

After determining res, a rule-based procedure is then used to generate specific scaling strategies based on $target$ and res. To be specific, more resources should be allocated to $target$ whenever $res > 0$. In particular, if the remaining CPU capacity $remain_vcpu$ of the VM hosting $target$ is larger than res, *vertical scaling* is applied to allocate res vCPUs to $target$; otherwise, $remain_vcpu$ vCPUs are allocated to $target$ and a new container is created by horizontal scaling. The newly created container is allocated $(res - remain_vcpu)$ vCPUs. On the other hand, resources allocated to $target$ will be reduced if $res < 0$. If res is less than the total vCPUs of $target$, the vCPUs of $target$ are reduced by res; otherwise, the container $target$ is deleted as a result of *horizontal scaling*. If $res = 0$, the resources allocated to $target$ remain unchanged.

Subsequently, the outputs of MLP_i and MLP_s are transformed into an autoscaling action $\langle target, scaling \rangle$.

4.3. Evolutionary Strategy-based Reinforcement Learning

In this paper, we employ ESRL (Salimans et al., 2017) to train the spatial-temporal autoscaling policy of *STAR*. The ESRL is a population-based algorithm to estimate the gradients of neural networks. Algorithm 1 presents the pseudo-code of the ESRL. The ESRL updates the parameters of the spatial-temporal autoscaling policy iteratively. The policy network first initials all trainable parameters $\hat{\theta} = \{\mathbf{W}_l, \vec{a}, \mathbf{W}_h, \mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v, \theta_s, \theta_g\}$. Each iteration starts with sampling N perturbations $[\epsilon_i]_{i=0,1,\ldots,N}$ from standard gaussian distribution $\mathcal{N}(0, 1)$ (line 4). Then, a population of N individuals $[\theta_i]_{i=0,1,\ldots,N}$ is generated (line 5), each individual is a set of parameters in the policy network.

The fitness of an individual θ_i (line 6) is calculated by

$$F(\theta_i) = -MRT(T) - \rho \cdot \max\big(0, (C(T) - budget(T))\big), \tag{12}$$

Algorithm 1 ESRL

Input: Population size: N, maximum epochs: max_gen, initial policy parameters: $\hat{\theta}$, learning rate: η, multi-variance gaussian noise standard deviation: σ

Output: Trained neural networks

1: $gen \leftarrow 0$
2: **while** $gen \leq max_gen$ **do**
3:     **for** $i = 0$ to N **do**
4:         Sample perturbation $\epsilon_i \sim \mathcal{N}(0, 1)$
5:         Update the policy π_i by using $\theta_i \leftarrow \hat{\theta} + \sigma\epsilon_i$
6:         Calculate Fitness $F(\theta_i)$ based on Eq. 12
7:     **end for**
8:     Estimate policy gradient $\nabla_\theta \mathbb{E}_{\epsilon_i \sim \mathcal{N}(0,1)} F(\hat{\theta} + \sigma\epsilon_i)$
9:     $\hat{\theta} \leftarrow \hat{\theta} + \sigma F(\hat{\theta} + \sigma\epsilon_i)$
10: **end while**

where ρ controls the intensity of the penalty for exceeding the *budget*.

Then, the parameters of the policy network are updated by the expectation of individuals' fitness (lines 8 and 9), which is the estimation of gradients, calculated by

$$
\begin{aligned}
\nabla_\theta \, \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)}\big[F(\hat{\theta} + \sigma\varepsilon)\big] &= \nabla_\theta \, \mathbb{E}_{x \sim \mathcal{N}(\hat{\theta}, \sigma^2 I)}[F(x)] = \mathbb{E}_x\big[F(x) \, \nabla_\theta \log \mathcal{N}(x; \hat{\theta}, \sigma^2 I)\big] \\
&= \frac{1}{\sigma^2} \, \mathbb{E}_x\big[F(x) \, (x - \hat{\theta})\big] = \frac{1}{\sigma} \, \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)}\big[F(\hat{\theta} + \sigma\varepsilon) \, \varepsilon\big] \\
&\approx \frac{1}{N\sigma} \sum_{i=1}^{N} F(\theta + \sigma\epsilon_i)\epsilon_i. \qquad (13)
\end{aligned}
$$

ESRL may require more training samples than some off-policy RL algorithms, e.g., TD3 and SAC, but the trade-off between sample efficiency and performance is acceptable. This is because *STAR* is trained offline using real-work historical traces in a simulated environment (Shi et al., 2023; Xu et al., 2022a; Xie et al., 2024; Liang et al., 2025; Bai et al., 2024), without interacting with the production system, resulting in negligible additional cost.

5. Performance Evaluation

In this section, we conduct comprehensive experiments to evaluate the performance of *STAR*. We first present the experiment settings, the *STAR* configurations and the competing approaches. Then, the experiment results and further analyses are presented.

5.1. Experiment Settings

For our experiments, we generate workloads for experiment from four real-world user request traces, NASA[4], Wiki (van Baaren, 2009) and Ali (Luo et al., 2022b). Fig. 9 illustrates these four trace patterns over a 960-time-unit period (equivalent to two days), where each time unit represents a 3-minute interval (Shi et al., 2021, 2020). The first 480 time units are used for training, while the remaining time units are used for testing. Autoscaling decisions are made every 3 minutes, in line with (Shi et al., 2023).

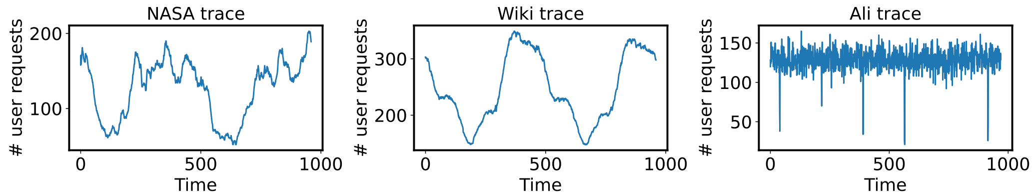

Fig. 9: Real-world user request traces

We evaluate three widely studied application architectures (Shi et al., 2020; Huang and Shen, 2015), as summarized in Fig. 10. Each application has a different number of services and structural configurations. For convenience, we refer to these applications as "A11", "A12", and "A13" based on their respective service counts. Five VM configurations from Amazon EC2[5] are equipped in the cloud environment of experiments. Table 2 provides details on the VM specifications. Meanwhile, a homogeneous PM configuration is used, where each PM has the same number of vCPU (i.e., 64) and memory capacity (i.e., 3200 GiB), following (Fang et al., 2025; Tan et al., 2020).

In total, we design nine experimental scenarios by combining three workload traces and three application architectures.

[4]https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html
[5]https://aws.amazon.com/ec2/pricing/on-demand/

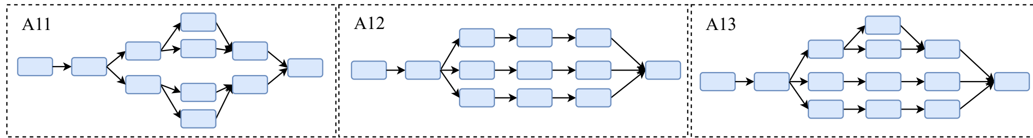

Fig. 10: The application architectures used in experiments

Table 2: VM types used in experiments.

VM Type	vCPU	Memory (GiB)	Hourly Price ($)
m5.xlarge	4	16	0.192
m5.2xlarge	8	32	0.384
m5.4xlarge	16	64	0.768
m5.8xlarge	32	128	1.536
m5.12xlarge	48	192	2.304

5.2. STAR *Configuration*

This paper sets the number of *graph attention layers* as 3, and the dimension of GAT's output is 64. The hidden dimension of each feedforward network is set as 64. The embedding dimension and output dimension of a self-attention layer are 16. The number of self-attention layers is 1.

For training, we set the population size of the ESRL as 40 (Huang et al., 2022). The maximum epoch is set as 1000. The learning rate η and the Gaussian noise standard deviation of the ESRL σ are set as 0.01 and 0.05, following the recommended settings in (Huang et al., 2022). Further analysis of different hyper-parameter settings, such as η and σ and the neural network architecture, is provided in Section 6.5. The *budget* of fitness function Eq. 12 is set as 200 USD per day (Cheng et al., 2024; Xie et al., 2024), while performances under different budgets are compared in Section 6.2. The penalty coefficient ϕ is set to 100, following (Shi et al., 2023). Different penalty coefficient settings are compared in Section 6.8.

5.3. Competing Approaches

We compare *STAR* with two heuristic-based autoscaling approaches and two state-of-the-art deep reinforcement learning (DRL)-based autoscaling approaches, as detailed below.

AWS-Scale (Amazon, 2022) is a threshold-based autoscaling strategy. Following the settings in (Shi et al., 2023; Nouri et al., 2019), we configure the CPU utilization thresholds for each container: an upper threshold of 0.8 and a lower threshold of 0.6. Specifically, when a container's CPU utilization surpasses the upper threshold, a new container is instantiated. Conversely, if the CPU utilization drops below the lower threshold, an existing container is removed.

ProScale (Cheng et al., 2023) is a heuristic-based proactive autoscaling approach that utilizes the SMA to predict future request workloads for each container. *Horizontal scaling* decisions are made based on the difference between the predicted future workload and the current request processing rate.

DeepScale (Shi et al., 2023) is a DQN-based autoscaling approach. It employs a deep neural network to make high-level decisions regarding resource allocation, including increasing, decreasing, or maintaining the number of provisioned containers. Subsequently, a heuristic strategy based on queueing theory is applied to determine low-level autoscaling actions, such as *horizontal scaling* and vertical scaling.

DRPC (Bai et al., 2024) is a distributed reinforcement learning approach for autoscaling. It first trains a central DRL agent using the TD3. Once the central DRL agent is trained, multiple deployment units are designed to imitate its decision-making process. These deployment units then execute autoscaling actions independently for each service in a distributed manner. Moreover, TD3 is utilized for periodically retraining deployment units during online interaction with the cloud environment.

5.4. Results

Fig. 11 compares the average optimization objective values in test traces over 10 independent runs for different methods across scenarios, where the objective is defined in Eq. (8). It can be observed that STAR consistently achieves the best objective values among all competing approaches, demonstrating its effective trade-off between QoS and cost.

Table 3 presents the detailed comparison of mean QoS performance (i.e., MRT) and violation rate (Vio) across test traces. We can observe from it that *STAR* outperforms the competing approaches in most scenarios except for W-11. In W-11, DeepScale outperforms *STAR* slightly. However, the DeepScale consumes significantly higher cost than *STAR* in this scenario, violating 34.75% of the given budget.

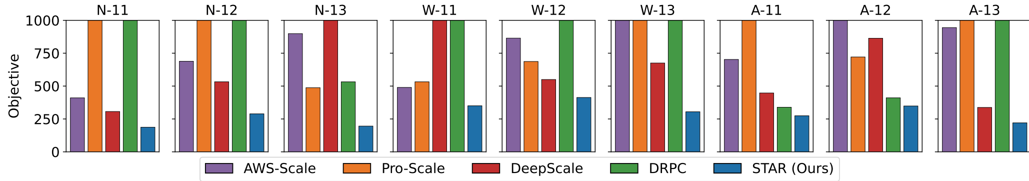

Fig. 11: Optimization objective (Eq. (8)) comparisons across different methods and scenarios.

More specifically, the "Vio" row in Table 3 demonstrates the performances of AWS-Scale, ProScale, DeepScale, DRPC and *STAR* in terms of cost, indicating the percentages of violating the budget. We can see that *STAR* consistently keeps the daily cloud provider costs below $200, whereas ProScale, DeepScale and DRPC exceed the budget in several scenarios, with the maximum violation reaching up to $67.10. AWS-Scale can always keep the cost of cloud providers below $200 per day. However, the MRT of AWS-Scale is significantly worse than *STAR*, as shown in Table 3.

Table 3: Performance comparisons in terms of MRT (ms) (defined in Eq. (4)) and cost violation degree "Vio" (defined in Eq. (7)) of cost exceeds the budget ($200 per day). "**Bold**": the best performances

Scenario	AWS-Scale		ProScale		DeepScale		DRPC		*STAR* (ours)	
	MRT	Vio	MRT	Vio	MRT	Vio	MRT	Vio	MRT	Vio
N-11	410.42	0	305.57	52.03	306.60	0	289.92	67.10	**187.15**	0
N-12	688.52	0	387.72	22.88	532.65	0	433.23	10.40	**288.85**	0
N-13	899.04	0	406.82	0.81	493.62	44.04	532.34	0	**195.76**	0
W-11	489.73	0	532.46	0	**318.29**	34.75	415.48	28.18	350.16	0
W-12	864.65	0	687.00	0	549.98	0	512.40	12.52	**413.26**	0
W-13	1080.44	0	482.13	13.18	675.37	0	491.68	56.17	**304.75**	0
A-11	702.46	0	680.03	8.65	447.48	0	339.05	0	**274.29**	0
A-12	1195.81	0	721.31	0	399.66	4.64	410.70	0	**348.62**	0
A-13	944.85	0	393.54	23.05	337.48	0	227.29	48.78	**220.32**	0

The above experiment results demonstrate that *STAR* can reduce mean response time by up to 78.23% (on N-13 the scenario), outperforming all competing approaches while ensuring the VM rental cost under $200 per day. Thanks to the design spatial-temporal autoscaling policy (see ablation results

in Section 6.4), *STAR* achieves strong generalization performance, effectively handling scenarios with different workload patterns and applications.

6. Further Analysis

We conduct further analysis in this section. Specifically, Section 6.2 presents performance comparisons under different budget settings. Section 6.3 reports experimental results across methods in terms of tail response time. Ablation studies and hyperparameter sensitivity analyses are shown in Section 6.4 and Section 6.5. Finally, Section 6.8 evaluates the performance of *STAR* under different penalty coefficients.

6.1. Generalization Ability Analysis

To validate the generalization ability of the autoscaling policy in *STAR*, we analyze the adaptability to application structure changes and long-term workload traces.

6.1.1. Adaptability to Application Structure Changes

To evaluate the adaptability of *STAR* to application structure changes, we apply the trained model to different application architectures. In particular, we train a model on A11 (referred to as *STAR-A11*) and evaluate it on the application structures of A12 and A13. Table 4 compares performance of *STAR-A11* with models trained on A12 and A13, i.e., *STAR-A12* and *STAR-A13*. The experimental results indicate that *STAR-A11* shows only negligible performance degradation compared to *STAR-A12* and *STAR-A13* on N-12 and N-13. Therefore, the trained *STAR* model can effectively adapt to application structure changes without retraining. This addresses an important practical challenge, as application structures may dynamically change in real-world environments.

Table 4: Comparison results of STAR-A11, STAR-A12 and STAR-A13 on N-12 and N-13, evaluated by the optimization objective in Eq. (8) (lower is better).

Scenario	STAR-A11	STAR-A12	STAR-A13
N-12	293.00 ± 40.08	288.85 ± 55.67	⟍
N-13	199.25 ± 46.58	⟍	195.76 ± 45.09

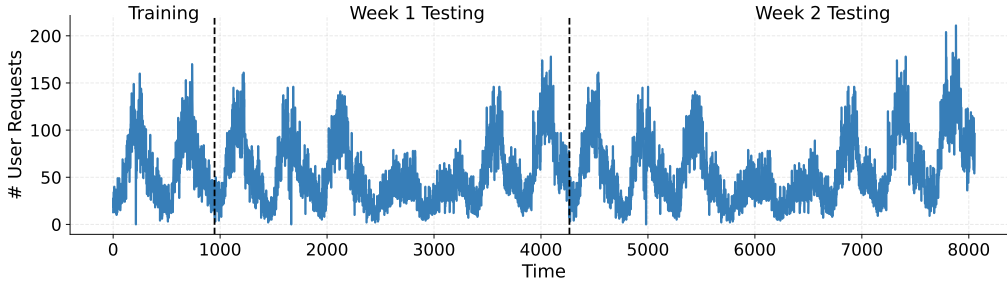

Fig. 12: 16 days workload traces of NASA.

6.1.2. Adaptability to Weekly Workload

We further evaluate the performance of *STAR* under a long-term weekly NASA workload trace[6], as shown in Fig. 12. Specifically, this trace covers 16 days of workload, where the first two days are used for training and the remaining 14 days (two weeks) are used for testing. The testing trace presents distinct workload patterns and covers a substantially longer duration compared to the training trace. The results in Table 5 show that *STAR* maintains stable MRT and cost across all 14 testing days under weekly workload patterns. In particular, similar performance is observed on the same weekdays across the two weeks, indicating that *STAR* effectively generalizes to weekly workload trends.

Table 5: Comparison of MRT and Cost for each day over two weeks.

Week	Mon		Tue		Wed		Thu		Fri		Sat		Sun	
	MRT	Cost	MRT	Cost	MRT	Cost	MRT	Cost	MRT	Cost	MRT	Cost	MRT	Cost
Week 1	118.42	124.84	114.75	154.33	73.17	175.72	69.05	182.02	63.58	182.02	74.91	182.02	79.18	182.02
Week 2	85.95	187.89	97.34	191.49	91.78	190.22	92.77	195.64	107.75	196.22	92.12	190.22	93.99	199.22

6.2. Performance Comparison with Different Budget

We compare the performances of *STAR* with stringent and relaxed cost budgets, which are \$150 and \$250, respectively, as shown in Table 6. Specifically, both stringent and relaxed budgets can be satisfied by *STAR*. Moreover, the *STAR* achieves lower MRT under \$250 budgets when compared to the results under \$150 budgets. The reason is that a relaxed budget allows for

[6]https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

the provision of more resources to the containers, which is aligned with the practice experiences.

Table 6: Performance comparisons of *STAR* with different budget: $150 and $250.

Scenario	$150		$250	
	MRT (ms)	cost ($)	MRT (ms)	cost ($)
N-11	201.4251	149.2694	177.0289	240.7732
N-12	295.9855	147.7732	280.5504	245.1824
N-13	199.1767	148.3648	177.1157	238.7432
W-11	389.2478	145.3542	311.5714	248.7425
W-12	412.5571	149.2874	382.2712	249.3781
W-13	355.2157	148.6587	255.7556	244.3125

6.3. Tail Response Time

The tail response time indicates the maximum response time under different percentages of user requests, which is another important metric of QoS. Fig. 13 presents the comparison of tail response time among STAR and competing approaches on N-13. This result shows that *STAR* achieve the least response time in all percentiles. Specifically, *STAR* enables 99% of user requests to be responded to within 521 milliseconds, significantly outperforming all other competing approaches.

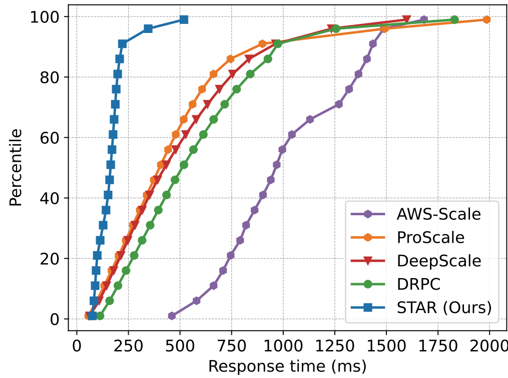

Fig. 13: Maximum response time under different percentages of user requests on N-13

6.4. Ablation Studies

To evaluate the effectiveness of key components in STAR, we conduct ablation studies on: (1) effectiveness of the spatial feature encoder, (2) effectiveness of the temporal feature encoder, (3) effectiveness of GAT, and (4) effectiveness of transformer.

6.4.1. Effectiveness of the Spatial and Temporal Feature Encoders

To evaluate the effectiveness of the spatial feature encoder, we remove it from *STAR*, called *w/o spatial*. Specifically, *w/o spatial* follows the design of MLP in DRPC (Bai et al., 2024) to capture features of the cloud environment and generate containers embeddings. Table 7 shows the performance comparisons of *STAR* and *w/o spatial* in terms of the optimization objective in Eq. (8). The results indicate that *w/o spatial* performs significantly worse than *STAR* in all of N-11, N-12 and N-13. These results verify the effectiveness of the spatial feature encoder to capture spatial dependencies of containers for autoscaling decisions.

Moreover, we also conduct experiments to test the effectiveness of the temporal feature encoder. We remove the temporal feature encoder component from *STAR*, giving rise to a variant, named *w/o temporal*. Table 7 demonstrates the comparison results of *w/o temporal* and *STAR* in terms of total reward. We can observe that the *w/o temporal* is significantly worse than *STAR* on N-11, N-12 and N-13, validating the effectiveness of the temporal feature encoder.

Table 7: Effectiveness of spatial feature encoder and temporal feature encoder, evaluated by the optimization objective in Eq. (8) (lower is better).

Scenario	w/o spatial	w/o temporal	*STAR*
N-11	247.38 ± 108.77	209.24 ± 33.54	197.68 ± 26.82
N-12	312.06 ± 55.23	266.73 ± 75.93	264.89 ± 55.67
N-13	355.25 ± 66.71	212.46 ± 26.88	186.10 ± 45.09

In summary, the ablation results show that removing either the spatial or temporal feature encoder significantly degrades the performance of *STAR* across N-11, N-12, and N-13. This demonstrates that both spatial dependencies among containers and temporal workload dynamics are critical for effective autoscaling decisions.

6.4.2. Effectiveness of GAT

To validate the effectiveness of the GAT in the spatial feature encoder, we replace it with a Graph Convolutional Network (GCN) (Kipf, 2016) and GraphSAGE (Hamilton et al., 2017), resulting in two variants, namely STAR-GCN and STAR-SAGE. Table 8 reports the comparison results among STAR-GCN, STAR-SAGE and STAR. It can be observed from this table that *STAR* is significantly better than STAR-GCN and STAR-SAGE, which demonstrates the effectiveness of the GAT in STAR. The superiority of GAT over GCN and GraphSAGE lies in its ability to learn adaptive attention weights over neighboring containers (Wang et al., 2025; Vrahatis et al., 2024), enabling context-aware aggregation that better captures heterogeneous dependencies for autoscaling decisions.

Table 8: Effectiveness of GAT in the spatial feature encoder, evaluated by the optimization objective in Eq. (8) (lower is better).

Scenario	STAR-GCN	STAR-SAGE	*STAR*
N-11	406.20 ± 275.33	228.02 ± 43.45	197.68 ± 26.82
N-12	794.50 ± 105.43	275.05 ± 55.60	264.89 ± 55.67
N-13	462.22 ± 35.27	189.92 ± 56.08	186.10 ± 45.09

6.4.3. Effectiveness of Transformer

To investigate the contribution of Transformer in the temproal feature encoder, we replace the Transformer with LSTM (Graves, 2012) and GRU (Chung et al., 2014), resulting in STAR-LSTM and STAR-GRU. As shown in Table 9, *STAR* significantly outperforms STAR-GCN and STAR-SAGE in all of N-11, N-12 and N-13. These results highlight the Transformer component can better capture temporal features than LSTM and GRU.

Table 9: Effectiveness of Transformer in the temporal feature encoder, evaluated by the optimization objective in Eq. (8) (lower is better).

Scenario	STAR-LSTM	STAR-GRU	*STAR*
N-11	245.10 ± 32.34	458.27 ± 196.41	197.68 ± 26.82
N-12	296.94 ± 27.18	327.80 ± 46.73	264.89 ± 55.67
N-13	194.44 ± 32.13	251.34 ± 116.13	186.10 ± 45.09

6.5. Hyper-Parameters Analysis

In this section, we investigate how the design of the neural architecture and the ESRL configuration influence the performance of STAR.

6.5.1. Neural Architecture Settings

To investigate the performance under different neural network architecture settings, we compare the performances of *STAR* in different settings. Specifically, we denote STAR-G2, STAR-G3, STAR-G4 and STAR-G5 as *STAR* with 2, 3, 4 and 5 graph attention layers. Meanwhile, we denote STAR-S1, STAR-S2, STAR-S3 and STAR-S4 as *STAR* with 1, 2, 3 and 4 self-attention layers.

Fig. 14 presents the average curves and the corresponding 95% confidence interval (CI) in different scenarios. Green curves in Fig. 14 demonstrate that STAR-G2 cannot effectively learn. This is because the limited number of GAT layers restricts the model's ability to capture sufficient structural and contextual information from multi-hop neighborhoods. Meanwhile, STAR-G4 and STAR-G5 exhibit higher instability (high 95% CI) than *STAR* on N-13. The instability observed when increasing the number of GAT layers might arise from the over-smoothing issue (Keriven, 2022), which causes the learned node embeddings to become overly similar and reduces the model's expressive power. In a word, *STAR* with 3 GAT layers achieves the best performance and the most stable training process.

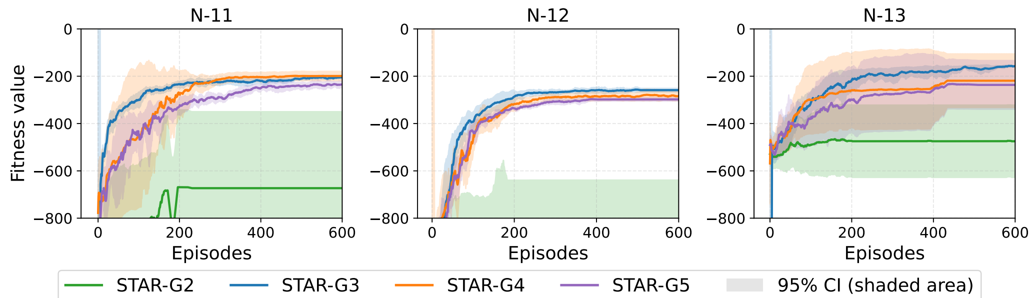

Fig. 14: Average fitness curves (Eq. (12)) under different numbers of graph attention layers

Fig. 15 shows the average curves and the 95% CI of STAR-S1, STAR-S2, STAR-S3, and STAR-S4 in different scenarios. The results indicate that STAR-S1 achieves the most stable and the best performance across all of N-11, N-12, and N-13. When stacking more self-attention layers (i.e., STAR-S2,

STAR-S3, STAR-S4), repeated global aggregation leads to over-smoothing of representations (Nguyen et al., 2023), which in turn diminishes the model's discriminative capability and training stability.

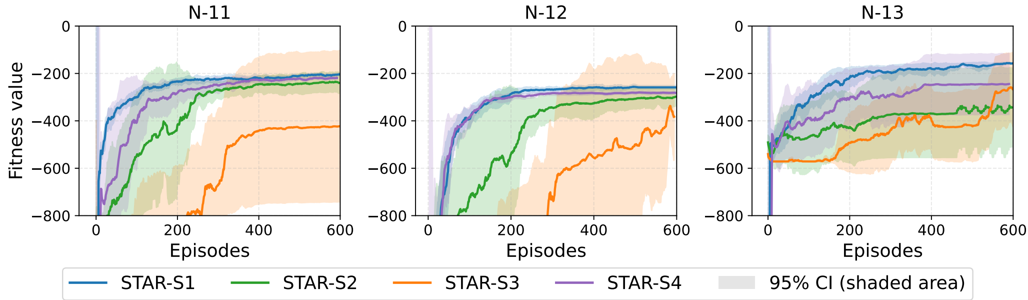

Fig. 15: Average fitness curves (Eq. (12)) under different number of self-attention layers

6.5.2. *ESRL Configuration*

Furthermore, we compare the performance of *STAR* under different settings of ESRL's hyperparameters. Particularly, we denote STAR-η0.01, STAR-η0.005, STAR-η0.001, and STAR-η0.0005 as *STAR* with learning rate η equal to 0.01, 0.005, 0.001, and 0.0005, respectively. Likewise, STAR-σ0.1, STAR-σ0.05, STAR-σ0.01, and STAR-σ0.005 refer to variants of *STAR* with Gaussian noise standard deviation σ set to 0.1, 0.05, 0.01, and 0.005, respectively. Moreover, STAR-pop20, STAR-pop30, STAR-pop40, STAR-pop50 and STAR-pop60 indicate the population sizes are set as 20, 30, 40, 50, and 60.

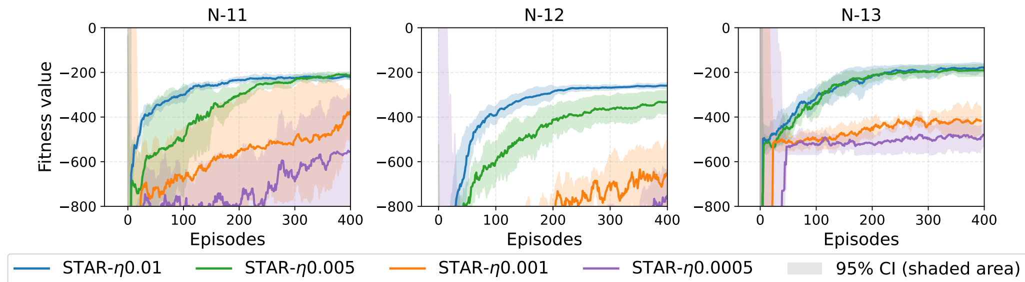

Fig. 16: Average fitness curves (Eq. (12)) under different settings of learning rate.

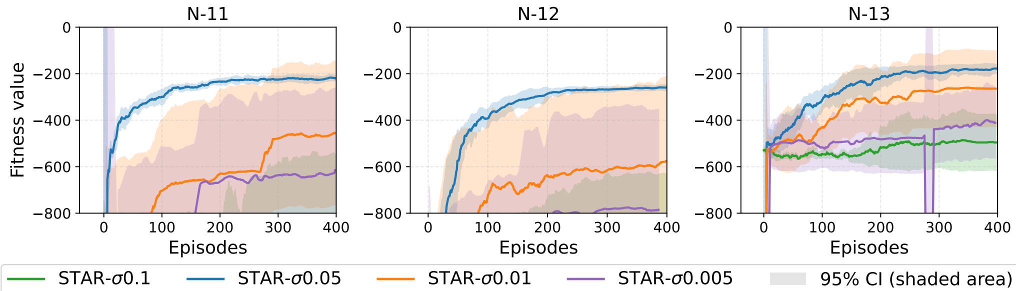

Fig. 17: Average fitness curves (Eq. (12)) under different Gaussian noise standard deviation.

Fig. 16 shows the comparison results under different learning rate η. It can be observed that STAR-η0.01 yields the most stable and efficient convergence, in contrast to $\eta \leq 0.001$, which fails to learn an effective policy within a reasonable number of iterations. Moreover, Fig. 17 compares the performances under different σ settings. The comparison results indicate that *STAR-σ0.05* achieves the best and most stable performance among all settings, while other configurations fail to converge effectively and show unstable learning behavior.

Fig. 18 demonstrates the convergence curves of *STAR* under different population sizes. We can observe from this figure that *STAR* cannot learn effectively under small population sizes (i.e., 20 and 30) due to the limited exploration capability. For population sizes of 40 or higher, *STAR* achieves comparable final performance across different settings, while larger populations result in a slightly faster convergence speed. Higher early-stage returns arise from more accurate gradient estimation with larger population sizes in ESRL. Specifically, a larger population size enhances the exploration dynamics, which in turn improves training stability. This leads to a faster convergence. Table 10 compares the ability of *STAR* to capture long-term workloads (one week traces in Fig. 12) under different population size settings.

The above results show that *STAR* achieves stable MRT and consistent average daily cost when the population size is 40 or higher, while smaller populations lead to degraded performance. A larger population size reduces sample efficiency, so we set the population size to 40 to trade off performance and sample efficiency.

In conclusion, based on existing studies (Huang et al., 2022; Shen et al.,

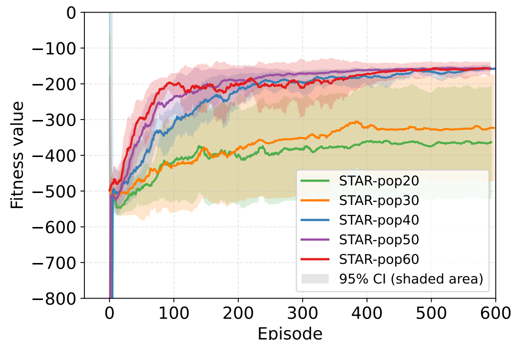

Fig. 18: The convergence curve of *STAR* under different population sizes on N-13. Each curve represents a different population size (20, 30, 40, 50, and 60).

Table 10: Comparison of STAR performance under a workload spanning one week with different population sizes, showing the impact on MRT and Average Daily Cost (ADC)

Metric	STAR-pop20	STAR-pop30	STAR-pop40	STAR-pop50	STAR-pop60
MRT	295.47	255.98	114.98	94.51	101.52
ADC	54.83	185.24	175.72	178.37	182.02

2024, 2025) and our experimental results, the tuned ESRL hyperparameters (i.e., $\eta = 0.01$, $\sigma = 0.05$, and population size 40) show strong robustness without the need for further fine-tuning across different problem instances. Empirical studies suggest that a learning rate in the range of $0.005 \leq \eta \leq 0.01$ and a Gaussian noise standard deviation around 0.05 are appropriate choices for cloud-related tasks (Huang et al., 2022; Shen et al., 2024, 2025). When the population size is set between 40 and 60, the ESRL exhibits similar performances.

6.6. Comparison with Other DRL Algorithm

To highlight the rationale for choosing ESRL, we compare the performance of *STAR* trained with PPO (Schulman et al., 2017) and ESRL, referred to as *STAR-PPO* and *STAR-ESRL*, respectively. PPO is a popular and effective on-policy reinforcement learning algorithm that has been widely applied across various domains (Yang et al., 2025; Song et al., 2022; Zhan et al., 2022). Fig. 19 presents the convergence curve of *STAR-PPO* and *STAR-ESRL* under the test scenario. We can observe from this figure that while *STAR-PPO* achieves slightly higher returns (without a statistically significant difference), *STAR-ESRL* exhibits substantially more stable per-

formance. Extensive practical evidence indicates that ESRL delivers more stable and superior performance than other DRL algorithms in cloud-related problems (Huang et al., 2022; Shen et al., 2024) as well as in other practical application domains (Zheng et al., 2025; Cai et al., 2025; Liu et al., 2024). Therefore, ESRL is a well-suited algorithm for solving autoscaling problems.

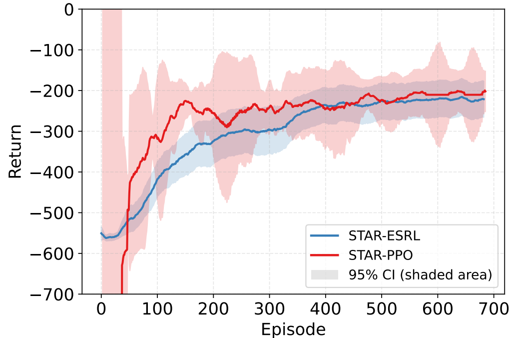

Fig. 19: Comparison results of STAR-PPO and STAR-ESRL

6.7. Efficiency Analysis

The training of *STAR* in this paper is equipped with 40 CPU cores based on AMD EPYC Milan architecture, and 40 GiB RAM[7]. For *STAR*, the average training time per epoch is approximately 1.5 minutes, with convergence achieved within 400 epochs in each scenario Moreover, *STAR* can make autoscaling decisions in an average of 2.15 microseconds, demonstrating extremely low latency. This means *STAR* is highly applicable for real-time autoscaling in production cloud environments, in which rapid adaptation to workload fluctuations is essential.

6.8. Performance Comparison with Different Penalty Coefficient

To examine the impact of the penalty coefficient in the fitness function (Eq. (12)), we evaluate the performance under different values of ρ. As shown in Table 11, no significant differences in ART are observed when ρ is set to 50, 100, 150, or 200, while all settings effectively enforce the budget constraint. This result indicates that the fitness function is not sensitive to

[7]https://www.nesi.org.nz/services/high-performance-computing-and-data-analytics/platforms

the specific choice of ρ, as cost is treated as a soft constraint rather than a competing optimization objective, and the learning process remains focused on optimizing MRT under the budget constraint.

Table 11: Performance comparison with penalty coefficient (ρ).

ρ	N-11		N-13	
	MRT (ms)	cost ($)	MRT (ms)	cost ($)
50	191.65	162.50	198.63	190.31
100	187.15	150.1	195.76	170.6
150	198.83	141.77	193.82	147.80
200	193.23	148.2	193.60	157.10

7. Conclusion and Future Work

To ensure the QoS performance of cloud applications while maintaining VM rental cost under a given budget, this paper proposes *STAR*, a novel DRL-based autoscaling approach. Specifically, a novel spatial-temporal encoding method is proposed to explicitly account for bi-directional spatial dependencies and the historical workload pattern of containers, leading to more effective autoscaling decisions than existing autoscaling approaches. Moreover, different from existing DRL-based autoscaling approaches, *STAR* enables fine-grained container-level autoscaling actions. Through extensive experiments using real-world user request traces, we demonstrate that *STAR* consistently outperforms state-of-the-art autoscaling approaches, including two threshold-based and two DRL-based approaches. In the future, we will investigate online DRL algorithms to periodically online fine-tune the policy over a fixed time horizon, which could further enhance adaptability in real cloud environments. Moreover, we plan to extend the framework to handle multiple containers concurrently within each autoscaling decision.

References

Amazon. AWS Auto Scaling, 2022. URL `https://aws.amazon.com/autoscaling/`. Accessed: 2024-11-21.

Jonatha Anselmi, Bruno Gaujal, and Louis-Sébastien Rebuffi. Non-stationary gradient descent for optimal auto-scaling in serverless platforms. *IEEE Transactions on Networking*, 2025.

Haoyu Bai, Minxian Xu, Kejiang Ye, Rajkumar Buyya, and Chengzhong Xu. Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. *IEEE Transactions on Services Computing*, 2024.

Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.

Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. KOSMOS: vertical and horizontal resource autoscaling for kubernetes. *ICSOC*, 13121:821–829, 2021.

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

B. Burns, J. Beda, and K. Hightower. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2019.

Qiuting Cai, Yiwen Zhang, Kaitong Zheng, Ya-Hui Jia, and Huaiguang Jiang. An evolutionary reinforcement learning method for multi-energy microgrid energy management. In *2025 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–4. IEEE, 2025.

Liao Chen, Shutian Luo, Chenyu Lin, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. Derm: Sla-aware resource management for highly dynamic microservices. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 424–436. IEEE, 2024.

Ke Cheng, Sheng Zhang, Chenghong Tu, Xiaohang Shi, Zhaoheng Yin, Sanglu Lu, Yu Liang, and Qing Gu. Proscale: Proactive autoscaling for microservice with time-varying workload at the edge. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

Ke Cheng, Sheng Zhang, Meizhao Liu, Yingcheng Gu, Liu Wei, Huanyu Cheng, Kai Liu, Yu Song, Xiaohang Shi, Andong Zhu, et al. Geoscale: Microservice autoscaling with cost budget in geo-distributed edge clouds. *IEEE Transactions on Parallel and Distributed Systems*, 35(4):646–662, 2024.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Noah E El-Zathry, Tarek El-Attar, Ibrahim Sabry, Rasheedat M Mahamood, Stephen Akinlabi, Wai Lok Woo, Esther Akinlabi, and Ahmed El-Assal. Fused deposition modeling component quality enhancement by experimental investigation and ann prediction. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, pages 1–24, 2025.

Zhengxin Fang, Hui Ma, Gang Chen, and Sven Hartmann. Energy-efficient and communication-aware resource allocation in container-based cloud with group genetic algorithm. In *International Conference on Service-Oriented Computing*, pages 212–226. Springer, 2023a.

Zhengxin Fang, Hui Ma, Gang Chen, and Sven Hartmann. A group genetic algorithm for energy-efficient resource allocation in container-based clouds with heterogeneous physical machines. In *Australasian Joint Conference on Artificial Intelligence*, pages 453–465. Springer, 2023b.

Zhengxin Fang, Hui Ma, Gang Chen, Sven Hartmann, and Shiping Chen. A communication-aware and energy-efficient genetic programming based method for dynamic resource allocation in clouds. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 421–436. Springer, 2025.

Zhengxin Fang, Hui Ma, Gang Chen, and Rajkumar Buyya. Hgraphscale: Hierarchical graph learning for autoscaling microservice applications in container-based cloud computing. *IEEE Transactions on Services Computing*, 2026.

FinOps, 2025. URL `https://www.finops.org/introduction/what-is-finops/`.

Yisel Garí, David A Monge, and Cristian Mateos. A q-learning approach for the autoscaling of scientific workflows in the cloud. *Future Generation Computer Systems*, 127:168–180, 2022.

Yisel Garí, Elina Pacini, Luciano Robino, Cristian Mateos, and David A Monge. Online rl-based cloud autoscaling for scientific workflows: Evalu-

ation of q-learning and sarsa. *Future Generation Computer Systems*, 157: 573–586, 2024.

Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Kuo-Chan Huang and Bo-Jun Shen. Service deployment strategies for efficient execution of composite saas applications on cloud platform. *Journal of Systems and Software*, 107:127–141, 2015.

Victoria Huang, Chen Wang, Hui Ma, Gang Chen, and Kameron Christopher. Cost-aware dynamic multi-workflow scheduling in cloud data center using evolutionary reinforcement learning. In *International Conference on Service-Oriented Computing*, pages 449–464. Springer, 2022.

Saikishor Jangiti, Varadarajan Vijayakumar, and V Subramaniyaswamy. Hybrid best-fit heuristic for energy efficient virtual machine placement in cloud data centers. *EAI Endorsed Transactions on Energy Web*, 7(26), 2020.

Nicolas Keriven. Not too little, not too much: a theoretical analysis of graph (over) smoothing. *Advances in Neural Information Processing Systems*, 35:2268–2281, 2022.

TN Kipf. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Lawrence Berkeley National Laboratory. Traces available in the internet traffic archive, 2022. URL `http://ita.ee.lbl.gov/html/traces.html`. Online.

Xin Li, Junsong Zhou, Xin Wei, Dawei Li, Zhuzhong Qian, Jie Wu, Xiaolin Qin, and Sanglu Lu. Topology-aware scheduling framework for microservice applications in cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1635–1649, 2023.

Pengjuan Liang, Yaling Xun, Jianghui Cai, and Haifeng Yang. Autoscaling of microservice resources based on dense connectivity spatio–temporal gnn and q-learning. *Future Generation Computer Systems*, page 107909, 2025.

Haiying Liu, Kun Wu, Kuihua Huang, Guangquan Cheng, Rui Wang, and Guohua Liu. Optimization of large-scale uav cluster confrontation game based on integrated evolution strategy. *Cluster Computing*, 27(1):515–529, 2024.

Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022a.

Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: Microservice auto scaling via workload learning. In *Proceedings of the ACM Symposium on Cloud Computing*, 2022b.

Shutian Luo, Chenyu Lin, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, Huanle Xu, and Chengzhong Xu. Optimizing resource management for shared microservices: a scalable system design. *ACM Transactions on Computer Systems*, 42(1-2):1–28, 2024.

Nima Mahmoudi and Hamzeh Khazaei. Performance modeling of metric-based serverless computing platforms. *IEEE Transactions on Cloud Computing*, 11(2):1899–1910, 2022.

Chunyang Meng, Jingwan Tong, Maolin Pan, and Yang Yu. Hra: An intelligent holistic resource autoscaling framework for multi-service applications. In *2022 IEEE International Conference on Web Services (ICWS)*, pages 129–139. IEEE, 2022.

Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–65. IEEE, 2023.

Microsoft, 2024. URL `https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/tradeoffs`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Tam Nguyen, Tan Nguyen, and Richard Baraniuk. Mitigating oversmoothing in transformers via regularized nonlocal functionals. *Advances in Neural Information Processing Systems*, 36:80233–80256, 2023.

Seyed Mohammad Reza Nouri, Han Li, Srikumar Venugopal, Wenxia Guo, MingYun He, and Wenhong Tian. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. *Future Generation Computer Systems*, 94:765–780, 2019.

Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graph neural network-based slo-aware proactive resource autoscaling framework for microservices. *IEEE/ACM Transactions on Networking*, 2024.

Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.

Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Hierarchical auto-scaling policies for data stream processing on heterogeneous resources. *ACM Transactions on Autonomous and Adaptive Systems*, 18 (4):1–44, 2023.

IBRAHEEM SABRY, AM El-Kassas, AM Khourshid, and HM Hindawy. Comparison of rsm and ra with ann in predicting mechanical properties of friction stir welded aluminum pipes. *Engineering and Technology in India*, 8(1and2):1–14, 2017.

Ibrahim Sabry. Exercising hybrid statistical tools ga-ann and ga-anfis to optimize underwater friction stir welding process parameters for tensile strength improvement. *EPPM 2021*, page 88, 2021.

Ibrahim Sabry, Abdel Hamid I Mourad, and Dinu Thomas Thekkuden. Comparison of mechanical characteristics of conventional and underwater friction stir welding of aa 6063 pipe joints. *Int. Rev. Mech. Eng*, 14(1):64–70, 2020.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ya Shen, Gang Chen, Hui Ma, and Mengjie Zhang. Cost-aware dynamic cloud workflow scheduling using self-attention and evolutionary reinforcement learning. In *International Conference on Service-Oriented Computing*, pages 3–18. Springer, 2024.

Ya Shen, Gang Chen, Hui Ma, and Mengjie Zhang. Gates: Cost-aware dynamic workflow scheduling via graph attention networks and evolution strategy. *arXiv preprint arXiv:2505.12355*, 2025.

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann. Location-aware and budget-constrained service deployment for composite applications in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1954–1969, 2020.

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann. Cost-effective web application replication and deployment in multi-cloud environment. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1982–1995, 2021.

Tao Shi, Hui Ma, Gang Chen, and Sven Hartmann. Auto-scaling containerized applications in geo-distributed clouds. *IEEE Transactions on Services Computing*, 2023.

Wen Song, Xinyang Chen, Qiqiang Li, and Zhiguang Cao. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 19(2):1600–1610, 2022.

Satish Narayana Srirama, Mainak Adhikari, and Souvik Paul. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications*, 160:102629, 2020.

Boxiong Tan, Hui Ma, Yi Mei, and Mengjie Zhang. A cooperative coevolution genetic programming hyper-heuristics approach for on-line resource

allocation in container-based clouds. *IEEE Transactions on Cloud Computing*, 10(3):1500–1514, 2020.

E.-J. van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. Master's thesis, VU Amsterdam, 2009. URL `http://www.wikibench.eu/wp-content/`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050 (20):10–48550, 2017.

Aristidis G Vrahatis, Konstantinos Lazaros, and Sotiris Kotsiantis. Graph attention networks: a comprehensive review of methods and applications. *Future Internet*, 16(9):318, 2024.

Anping Wan, Qing Chang, AL-Bukhaiti Khalil, and Jiabo He. Short-term power load forecasting for combined heat and power using cnn-lstm enhanced by attention mechanism. *Energy*, 282:128274, 2023.

Sheng Wang, Zhijun Ding, and Changjun Jiang. Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):98–115, 2020.

Yucheng Wang, Min Wu, Xiaoli Li, Lihua Xie, and Zhenghua Chen. A survey on graph neural networks for remaining useful life prediction: Methodologies, evaluation and future trends. *Mechanical Systems and Signal Processing*, 229:112449, 2025.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Linfeng Wen, Minxian Xu, Sukhpal Singh Gill, Muhammad Hilman, Satish Narayana Srirama, Kejiang Ye, and Chengzhong Xu. Statuscale: Status-aware and elastic scaling strategy for microservice applications. *ACM Transactions on Autonomous and Adaptive Systems*, 20(1):1–25, 2025.

Shuli Wen, Yu Wang, Yi Tang, Yan Xu, Pengfei Li, and Tianyang Zhao. Real-time identification of power fluctuations based on lstm recurrent neural network: A case study on singapore power system. *IEEE Transactions on Industrial Informatics*, 15(9):5266–5275, 2019.

Zhaojie Wen, Qiong Chen, Quanfeng Deng, Yipei Niu, Zhen Song, and Fangming Liu. Combofunc: joint resource combination and container placement for serverless function scaling with heterogeneous container. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

Shuaiyu Xie, Jian Wang, Bing Li, Zekun Zhang, Duantengchuan Li, and Patrick CK Hung. Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications. *IEEE Transactions on Services Computing*, 2024.

Mingzhe Xing, Hangyu Mao, and Zhen Xiao. Fast and fine-grained autoscaler for streaming jobs with reinforcement learning. In *IJCAI*, pages 564–570, 2022.

Minxian Xu, Chenghao Song, Shashikant Ilager, Sukhpal Singh Gill, Juanjuan Zhao, Kejiang Ye, and Chengzhong Xu. Coscal: Multifaceted scaling of microservices with reinforcement learning. *IEEE Transactions on Network and Service Management*, 19(4):3995–4009, 2022a.

Minxian Xu, Chenghao Song, Huaming Wu, Sukhpal Singh Gill, Kejiang Ye, and Chengzhong Xu. esdnn: deep neural network based multivariate workload prediction in cloud computing environments. *ACM Transactions on Internet Technology (TOIT)*, 22(3):1–24, 2022b.

Yifan Yang, Gang Chen, Hui Ma, and Mengjie Zhang. Dual-tree genetic programming for deadline-constrained dynamic workflow scheduling in cloud. In *International Conference on Service-Oriented Computing*, pages 433–448. Springer, 2022.

Yifan Yang, Gang Chen, Hui Ma, Cong Zhang, Zhiguang Cao, and Mengjie Zhang. Graph assisted offline-online deep reinforcement learning for dynamic workflow scheduling. In *The Thirteenth International Conference on Learning Representations*, 2025.

Guang Zhan, Xinmiao Zhang, Zhongchao Li, Lin Xu, Deyun Zhou, and Zhen Yang. Multiple-uav reinforcement learning algorithm based on improved ppo in ray framework. *Drones*, 6(7):166, 2022.

Shubo Zhang, Tianyang Wu, Maolin Pan, Chaomeng Zhang, and Yang Yu. A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning. In *2020 IEEE international conference on web services (ICWS)*, pages 489–497. IEEE, 2020.

Kaitong Zheng, Ya-Hui Jia, Kejiang Ye, and Wei-Neng Chen. Strategic evolutionary reinforcement learning with operator selection and experience filter. *IEEE Transactions on Neural Networks and Learning Systems*, 2025.