A Retrieval-Augmented Generation-driven Auto-scaling Method for Microservices in Cloud-native Environments

Xu Jiang*, Zhicheng Cai†, Lei Xu†, Xiaoping Li‡ and Rajkumar Buyya§
*School of Cyber Science and Engineering, Nanjing University of Science and Technology, Nanjing, China
†School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China
‡School of Computer Science and Engineering, Guangdong University of Technology, GuangDong, China
§School of Computing and Information Systems, University of Melbourne, Melbourne, Australia
First Corresponding Author: Zhicheng Cai (caizhicheng@njust.edu.cn)

***Abstract*—With the evolution of cloud-native technologies, auto-scaling of microservices has become essential for ensuring Quality of Service (QoS) and optimizing resource utilization. However, existing resource management methods encounter significant robustness challenges in complex, heterogeneous cloud environments. This paper proposes a Retrieval-Augmented Generation (RAG)-driven Auto-scaling Method (R-AM) for microservices. To enhance the robustness of the decision-making architecture and improve the interpretability of decision-processing, a RAG-based Decision Reasoning (RDR) method is proposed. This approach not only maps monitoring metrics onto semantic descriptions but also constructs an auto-scaling reasoning chains by integrating expert knowledge through RAG. Building upon the established reasoning chains, a Heterogeneous Tool Invoking (HTI) method is developed to enhance the execution capability of Large Language Models (LLMs) in the field of auto-scaling. The LLM acts as a central scheduler invoking specific heterogeneous tools to adjust the system based on the reasoning chains. To ensure that R-AM has continuous learning capabilities, we propose a System Feedback based Interactive Learning (SFIL) method. By maintaining continuous interaction with the system, the approach collects real-time execution outcomes and performance metrics, which serve as the basis for iteratively fine-tuning the scheduling logic of LLM. This data-driven evolution mechanism enables the R-AM to sensitively capture and adapt to dynamic workloads. Compared with state-of-the-art baselines, experimental results on a real kubernetes cluster show that R-AM reduces the Service Level Agreement (SLA) violation ratio of the 95th percentile response time from 11.17% to 6.7% while enhancing the success rate by 73.91%.**

***Index Terms*—Large Language Models, Microservices, Auto-scaling, Explainability, Robustness**

I. Introduction

With rapid advances in cloud computing, cloud-native applications have evolved into highly modular microservices architectures [1]. As a lightweight platform for these applications, containers achieve high decoupling between application logic and underlying infrastructure by providing standardized encapsulation of microservice functions and their runtime environments. This paradigm provides the technical foundation for cloud deployment and collaboration between microservices. However, within container clusters of fixed capacity, dynamically fluctuating workloads usually lead to an imbalance between the supply and demand of computing resources. Resource contention during peak periods triggers service delay variation and throughput degradation, whereas resource redundancy during off-peak hours results in significant computational waste and cost overruns. To guarantee SLAs and optimize resource allocation under dynamic workloads, container auto-scaling has emerged as a core task of cloud management systems [2], [3]. Traditional auto-scaling methods, such as heuristic rules (e.g., Kubernetes HPA), control theory (e.g., queuing models), and artificial intelligence (e.g., reinforcement learning), demonstrate superior performance in specific scenarios. However, in environments with large-scale deployment of microservices, these methods still face challenges such as lack of interpretability and limited generalization, making it difficult to achieve robust policy migration in highly dynamic cloud settings.

The first challenge is that environmental changes degrade the robustness of auto-scaling methods [4]. Due to microservice scale, call chain depths, and load characteristics significantly across business scenarios, methods are often deeply coupled with specific distributions of monitoring metrics. When the system environment changes or new business logic is deployed, these methods struggle to adapt to new environments through parameter fine-tuning or transfer learning, leading to a sharp degradation of generalization performance across heterogeneous tasks. Such highly coupled architectures significantly hinder the widespread adoption of auto-scaling technologies in large-scale cloud-native environments. Consequently, constrained by overfitting to scenario-specific data, traditional auto-scaling methods fail to achieve robust policy migration in highly dynamic and heterogeneous cloud-native scenarios.

Meanwhile, the second challenge is that the lack of interpretability in auto-scaling processes hinders the practical deployment of these methods in production [5]. Existing methods, such as those based on Deep Reinforcement Learning (DRL), are often characterized as black-box systems because their resulting scheduling actions lack explicit logical support. When confronted with abnormal workload fluctuations or

sudden performance bottlenecks, Operations and Maintenance (OM) engineers usually face difficulties in comprehend the empirical evidence or metric weightings behind specific scaling actions. Consequently, constrained by inherent nature of numerical mapping (e.g., the direct mapping mechanism in DRL), traditional auto-scaling methods fail to provide comprehensible semantic logic to rationalize complex auto-scaling behaviors.

In this paper, a RAG-driven Auto-scaling Method (R-AM) is proposed. LLM not only possesses logical reasoning capabilities but also integrates expert knowledge through RAG. Unlike traditional auto-scaling methods, R-AM converts data into semantic descriptions, decoupling the decision-making process from specific numerical distributions, and is able to reason about problems and generate interpretable reasoning chains. The primary contributions of this paper are summarized as follows:

1) A RAG-based Decision Reasoning (RDR) method is proposed to achieve decoupling between decision logic and metric values. By mapping monitoring metrics onto semantic descriptions and integrating an expert knowledge base, RDR establishes a reasoning chain from semantic descriptions to tool selection and parameter generation, thereby enhancing the robustness and explainability of the decision-making process.
2) A Heterogeneous Tool Invoking (HTI) method is proposed to to enhance the OM capability of LLM. By dynamically invoking auto-scaling algorithms within the tool library, this method enables LLMs to execute container auto-scaling tasks across complex environments.
3) A System Feedback based Interactive Learning (SFIL) method is proposed to continuously optimize the LLM. By leveraging real-time feedback from kubernetes, SFIL dynamically updates its local empirical knowledge base. This enables online self-iteration of policies and effectively overcomes the dependence of traditional LLMs on manual fine-tuning.

The rest of the paper is organized as follows. Section 2 reviews related works. The problems are described in Section 3. Section 4 introduces the proposed algorithms and Section 5 shows experimental results. Conclusions and future work are depicted in Section 6.

II. Related Work

This section presents the evolution of auto-scaling technologies within microservice architectures. the research focus has shifted from simplistic threshold-based triggers to complex neural-network-driven decision-making processes.

A. Traditional Rule-Driven Methods

Traditional auto-scaling algorithms are primarily driven by heuristic rules [6]–[8] and queuing models [9]–[11]. Threshold-based heuristics are widely deployed in open-source orchestration systems and commercial cloud services [6]. However, fixed thresholds struggle to adapt to the inherent dynamism of cloud environments and frequently trigger system oscillations [7]. To address this, Trihinas et al. [8] proposed the AdaFrame framework, which mitigates resource fluctuations by dynamically adjusting decision-window sensitivity based on continuous data-stream variations. Despite these improvements, the complex, non-linear, and non-constant coupling between physical metrics (e.g., CPU utilization) and QoS metrics makes it difficult for pure threshold mechanisms to achieve precise performance tracking. To enable proactive scaling, researchers have introduced queuing theory [22] (e.g., $M/M/N$, $G/G/1$) to model the mapping between request arrival rates and response time constraints [9]–[11]. Considering the intricate dependencies between microservices, Gias et al. [9] formulated the scaling problem as a non-linear optimization task, utilizing Layered Queuing Networks to solve for the optimal number of replicas in real-time. Nevertheless, the practical application of queuing models is often limited by the accuracy of parameter estimation. Cai et al. [10] introduced a feedback mechanism to enhance model robustness against parameter fluctuations. Furthermore, to address the limitations of the stationarity assumption in queuing theory, Lei et al. [11] proposed a hybrid control approach integrating state-space models with queuing networks, resolving container provisioning challenges under non-steady states through dynamic policy switching. Although theoretically rigorous, queuing models often lack deep perception of heterogeneous service logic when dealing with dynamic, mesh-like microservice topologies, making topology-aware adaptation difficult to achieve.

B. Machine Learning for Resource Scheduling

To overcome the limitations of static rules, Machine Learning (ML) has been widely applied to capture the complex dynamic characteristics of microservice workloads [12]–[14]. Addressing the challenges of non-stationary workloads and strong coupling among multi-dimensional resource metrics in cloud environments, Xu et al. [12] proposed an encapsulated spatial deep neural network framework. By mining the interdependencies between resources, this approach breaks the constraints of traditional univariate prediction models. However, traditional models often suffer from precision degradation in long-term forecasting due to cumulative errors. Zhao et al. [13] designed the MSCNet architecture, which employs multi-scale modeling to perform feature decomposition on raw workloads, effectively extracting both microscopic fluctuations and macroscopic trends. Despite the continuous evolution of prediction algorithms, they remain overly dependent on historical distributions and lack robustness when facing sudden anomalies or distribution shifts [14]. Another research paradigm is represented by end-to-end decision models based on Deep Reinforcement Learning (DRL) [15]–[18]. Qiu et al. [16] proposed the FIRM framework, which proves that the combination of data-driven and hardware-aware methods is an effective way to ensure the performance of complex microservices. However, the algorithms in FIRM need to rely on specific hardware characteristics. This means that some core functions of FIRM may not be fully enabled on different cloud vendors or old devices, and there are hardware

TABLE I
COMPARISON OF R-AM WITH STATE-OF-THE-ART AUTO-SCALING METHODS

Methods	Category	Provisioning Strategies	Interpretability	Robustness	Domain Knowledge
[6]–[8]	Rule-driven	Threshold	High	Low	Static experience
[9]–[11]	Rule-driven	Queuing Theory	High	Low	Mathematical priors
[12]–[14]	ML	ML	Low (black box)	Middle (fitting ability)	Feature Knowledge
[15]–[18]	ML	DRL	Low (black box)	Middle (fitting ability)	Policy Knowledge
[19], [20]	LLM	LLM	Middle (in-context)	High (reasoning ability)	General Knowledge
[21]	LLM	LLM+DRL	Middle (exist DRL)	Middle (exist DRL)	Policy Knowledge General Knowledge RAG
Our approach	LLM	LLM	High (reasoning chain)	High (metrics decoupling)	General Knowledge Static experience Mathematical priors

compatibility limitations. On the other hand, to tackle the state-space explosion problem in large-scale microservice clusters, Bai et al. [17] proposed a distributed reinforcement learning scheme. This approach decentralizes decision logic through knowledge distillation, achieving efficient resource allocation. Furthermore, to mitigate performance losses caused by container startup latency, Zou et al. [18] introduced the ReflexPilot architecture. By incorporating startup-overhead awareness to optimize scaling timing, it effectively alleviates SLA violations triggered by frequent cold starts. Nevertheless, while DRL excels in complex decision-making tasks, its black-box execution logic lacks intuitive interpretability, which significantly limits the reliability and trustworthiness of such algorithms.

C. LLM-driven Operations and Intelligence

The rise of LLMs has introduced a new paradigm of cognitive reasoning to AIOps [19]–[21]. Addressing the bottleneck of poor generalization exhibited by traditional deep learning models in heterogeneous environments, Xu et al. [19] proposed the UniLog framework. By incorporating similar cases as in-context prompts, UniLog achieves cross-project log pattern recognition and prediction without requiring parameter fine-tuning. To further enhance fault response efficiency in complex environments, researchers have begun exploring the synergy between LLMs and traditional algorithms. For instance, Wang et al. [20] constructed a tool-augmented multi-agent system that realizes an automated, full-closed-loop workflow from alert reception to root cause diagnosis. Similarly, Chen et al. [21] combined LLMs with Graph Reinforcement Learning, utilizing RL for large-scale policy filtering while leveraging LLMs to provide expert-level, semantic recovery suggestions based on topology graph states. Despite these advancements, the direct application of LLMs to real-time, high-frequency, and fine-grained container auto-scaling decisions remains in its infancy. Existing LLM-based O&M tools act primarily as offline consultants. Their decision logic is often decoupled from the underlying resource orchestration engines, lacking real-time in-the-loop interactive feedback mechanisms and performance guarantee constraints.

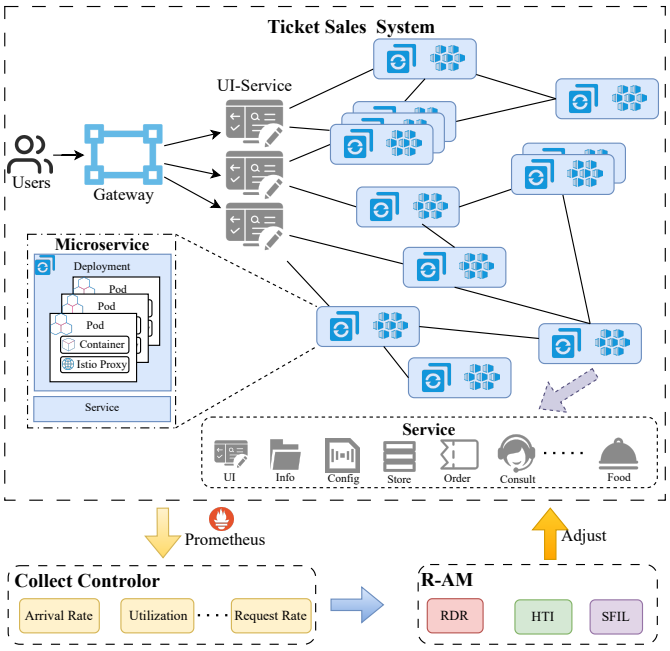

Fig. 1. Architecture of the considered cloud-native applications

The comparison of our approach R-AM with state-of-the-art methods shown in Table I. Existing methods for microservice auto-scaling continue to face critical challenges, including a lack of semantic awareness, black-box decision-making. The R-AM proposed in this paper constructs the decision-making reasoning chain through RDR, enhances the execution capability of LLMs through HTI, and establishes a learning cycle based on system feedback using SFIL.

III. PROBLEM DESCRIPTION

In cloud-native environments, applications are characterized by high heterogeneity and modularity. As shown in Fig. 1, applications are deployed on container orchestration platforms such as Kubernetes, composed of multiple microservices. Each microservice represents an independent functional unit within the application, and serves as a shared node across multiple

TABLE II
A SUMMARY OF FREQUENTLY USED NOTATIONS IN R-AM

Notations	Descriptions
M_t	the microservice metric set of the t-th decision cycle
S_t	the semantic descriptions of the t-th decision cycle
H_t	the temporal semantic states of the t-th decision cycle
sf_t	the sensitivity factors of the t-th decision cycle
rc_t	the reasoning chain of the t-th decision cycle
$\mathcal{D}$	the experience replay buffer of samples
$N_{\mathcal{D}}$	the threshold of accumulated samples

business service chains. The platforms receive user requests via API gateways, and distribute them to instances (pods) of the target microservice through load-balancing strategies. Within the cloud platform, the pod (container) is the atomic unit of resource allocation and scheduling, whose computational capacity is determined by the encapsulated container image and its runtime environment. However, microservices with different functionalities demonstrate significant variations in resource sensitivity. For example, I/O-intensive database query microservices and CPU-intensive numerical computation microservices present different resource bottleneck characteristics and latency distributions when subjected to identical workloads.

The objective of this paper is to design a container auto-scaling policy that minimizes leasing costs of containers while satisfying the response time SLA constraint SLA_{rt}. The SLA_{rt} dictates that the proportion V_{rt} of response times exceeding a threshold upper bound RT^{upper} remain below a specific threshold V_{rt}^{sla} (e.g., 10%). The SLA_{rt} is critical to maintaining the stability of the entire cloud-native application, which is the performance manifestation that users care about the most. Let q_i^t be the number of containers allocated to microservice s_i during the t-th decision cycle d_c, ρ_i represent the leasing price of a single container per second, and n be the total number of microservices. Each d_c has a duration of Δt. Accordingly, the optimization objective of the system is to minimize the cumulative cost over a time horizon T:

$$\min \mathcal{J} = \sum_{t=1}^{T} \sum_{i=1}^{n} \rho_i \cdot q_i^t \cdot \Delta t \tag{1}$$

Furthermore, this optimization problem is subject to the following resource boundary constraints:

$$q_{min} \leq \sum_{i=1}^{n} q_i^t \leq q_{max}, \quad \forall t \tag{2}$$

$$V_{rt} < V_{rt}^{sla} \tag{3}$$

where q_{min} and q_{max} represent the lower and upper bounds of the allowable container count within the cluster, respectively. These constraints ensure that resource allocation remains within physically feasible limits while guaranteeing that the QoS consistently meets user requirements. Frequently used notations are shown in Table II.

IV. PROPOSED CONTAINER AUTO-SCALING METHOD

In cloud-native microservice environments, a complex nonlinear dependencies exists between QoS (e.g., response time) and resource metrics (e.g., CPU and memory utilization). Traditional auto-scaling methods primarily rely on fixed metrics. However, the dynamic microservice scale and fluctuating workloads cause monitoring metrics to exhibit significant non-stationarity and uncertainty. To address this challenge and enhance the robustness, a RAG-based Decision Reasoning (RDR) method is proposed. RDR employs a semantic mapping mechanism to transform multi-dimensional monitoring metrics onto semantic descriptions. Subsequently, based on expert knowledge retrieved via RAG, the method constructs a reasoning chain from semantic descriptions to auto-scaling tool selection and parameter generation. This approach enables the generation of more interpretable resource auto-scaling action driven by the LLM. Meanwhile, building upon the established reasoning chains, a Heterogeneous Tool Invoking (HTI) method is developed to enhance the execution capability of LLMs in the field of auto-scaling. This approach empowers the LLM to function as an agent, invoking specialized scaling tools based on the reasoned context. By decoupling decision logic from physical execution, HTI ensures that the R-AM is able to leverage a diverse array of tools to perform precise resource adjustments. However, the microservice scale, business logic, and the performance characteristics of the underlying infrastructure undergo dynamic evolution over time. Even for LLMs with robust reasoning capabilities, the predefined knowledge bases of LLMs may become obsolete when faced with environment changes. To enable the R-AM to adapt to dynamic microservice scale and workload, this paper proposes a System Feedback-based Interactive Learning (SFIL) method. This method quantifies decision utility through a real-time feedback loop, driving the continuous policy optimization of the LLM. As shown in Fig. 2, the integrated approach encompassing RDR, HTI, and SFIL is designated as the RAG-driven Auto-scaling Method (R-AM). As described in Algorithm 1, within each d_c, R-AM collects microservice metrics M_t, which are then processed by RDR to generate rc_t. Subsequently, HTI invokes the appropriate auto-scaling method tm_t to adjust the number of container replicas. Concurrently, R-AM obtains system feedback through the reward function. When the number of samples in the experience replay buffer $\mathcal{D}$ more than the threshold $N_{\mathcal{D}}$, SFIL is activated to ensure the adaptive adjustment of the strategy.

A. RAG-based Decision Reasoning Method

To reduce the impact of metric variations on policy generalization across heterogeneous microservice architectures, RDR employs a transformation mechanism from numerical space to semantic space, coupled with RAG to construct reasoning chains.

1) Semantic State Modeling: Let $M_t = \{m_{i,t}\}_{i=1}^{n}$ denote the n-dimensional raw metrics collected at time t, including key metrics such as average response time RT_{avg}, 95th percentile response time RT_{95}, CPU utilization $cpu_{\%}$, memory

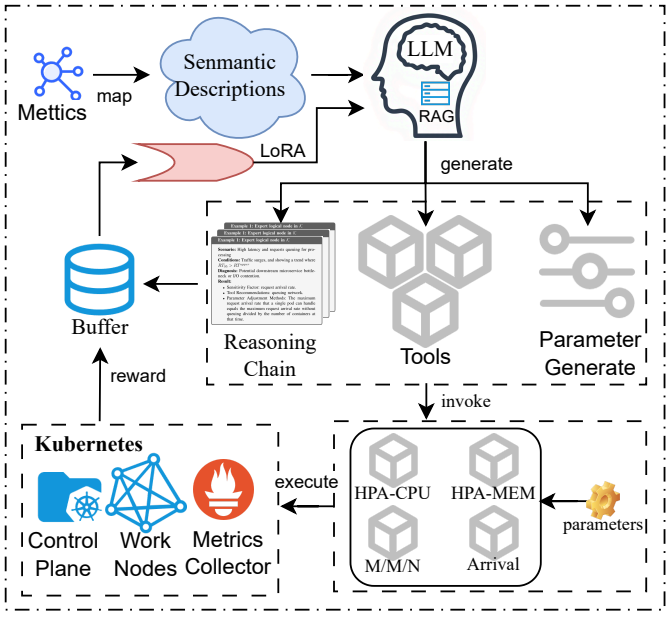

Fig. 2. Architecture of R-AM

Algorithm 1 R-AM: RAG-driven Auto-scaling Method

Input: d_c, $N_{\mathcal{D}}$

while True **do**
  Initialize Experience Replay Buffer $\mathcal{D} \leftarrow \emptyset$
  for each decision cycle d_c **do**
    Collect microservice metrics M_t;
    Generate reasoning chain $rc_t \leftarrow$ RDR;
    Invoke Tool $tm_t \leftarrow$ HTI;
    Adjust container number of microservices;
    Calculate reward of action;
    if $|\mathcal{D}| \geq N_{\mathcal{D}}$ **then**
      LoRA fine-tuning LLM $\leftarrow$ SFIL;
    end if
  end for
end while

usage $memory_{\%}$, request arrival rate λ, and request queue length. RDR utilizes a mapping operator to convert current metric values onto semantic descriptions S_t:

$$\mathcal{F} : M_t \rightarrow S_t \tag{4}$$

where the values of the metrics are retained, and then predefined statements are used to enrich the description. Fig. 3 shows the semantic mapping framework. This process not only preserves the numerical precision but also reduces the cognitive load of LLMs when understanding physical measurements.

2) RAG: RDR utilizes the temporal semantic state $H_t = \{S_{t-k}, \ldots, S_t\}$ as the query input to perform multi-branch parallel retrieval across multiple independent expert knowledge bases $\mathcal{K}$. The suggestion $\mathbb{H}_t$ obtained by retrieval:

$$\mathbb{H}_t = \text{Retrieve}(H_t, \mathcal{K}) \tag{5}$$

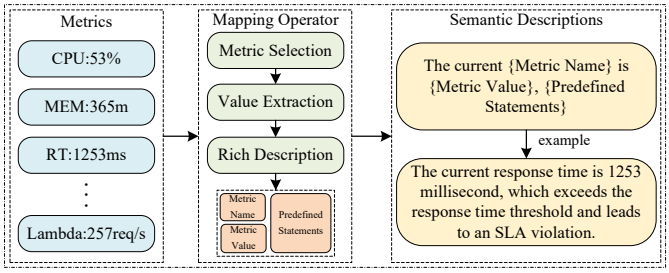

Fig. 3. The semantic mapping framework

Example 1: Expert logical node in $\mathcal{K}$

Scenario: High latency and requests queuing for processing
Conditions: Traffic surges, and showing a trend where $RT_{95} > RT^{upper}$
Diagnosis: Potential downstream microservice bottleneck or I/O contention.
Result:
- Sensitivity Factor: request arrival rate.
- Tool Recommendations: queuing network.
- Parameter Adjustment Methods: The maximum request arrival rate that a single pod can handle equals the maximum request arrival rate without queuing divided by the number of containers at that time.

Fig. 4. An example of decision rules for RAG

where the retrieval operator Retrieve$(\cdot)$ is implemented using FAISS. For example, this paper constructs independent $\mathcal{K}$ for scenarios such as CPU resource bottlenecks, memory resource constraints and traffic surges, respectively. Each decision rule within $\mathcal{K}$ contains diagnostic logic, sensitivity detection, tool recommendations, and parameter adjustment methods, providing knowledge units for decision reasoning. An example of such a decision rule is illustrated in Fig. 4.

3) Reasoning Chain Generation: The LLM receives an input tuple consisting of H_t, $\mathbb{H}_t$, and structured prompts, subsequently executing the following reasoning path to generate a rc_t:

Trend Analysis: Analyzes the trajectory of response times to identify risks of exceeding the upper threshold RT^{upper}.

Sensitivity Detection: Based on RAG-provided sensitivity criteria (e.g., observing response time fluctuations while CPU utilization remains constant), the model matches the scenario where states are located and detects sensitive factor sf_t.

Tool Selection and Parameter Generation: Recommend the auto-scaling tool based on the identified factors (e.g., invoking load-balancing optimization tools during high-concurrency rather than simple vertical scaling). Meanwhile, according to the parameter adjustment

methods in RAG, the LLM generates the key parameters p_t (e.g., CPU quotas or Pod request limits) corresponding to the auto-scaling tools.

Safety Filtering Mechanism: If the evaluation indicates that the system is operating within safety margins, the reasoning chain is automatically terminated to prevent unnecessary resource oscillations.

B. Heterogeneous Tool Invoking Method

Based on the rc_t derived from RDR, this paper relies on HTI to implement decision execution. HTI not only enhance the execution capability of LLM by expanding the heterogeneous tool library $\mathcal{TM}$, but also ensure the robustness in dynamic environments through strict bound protection mechanisms. In HTI, LLMs flexibly configure and orchestrate tools from $\mathcal{TM}$ to adapt to various auto-scaling requirements in dynamic environments.

1) Heterogeneous Tool Library: In this paper, the $\mathcal{TM}$ is equipped with three types of tools for LLM invocation, including the heuristic auto-scaling tool T_{HPA}, the queuing analysis tool $T_{Analytical}$, and the arrival rate toolT_{λ}.

Based on classic threshold-trigger mechanisms, T_{HPA} is ideal for scenarios where CPU or memory utilization acts as the primary bottleneck. The target replica count is calculated as:

$$q_t = \lceil q_{t-1} \times \frac{v_{current}}{v_{target}} \rceil \tag{6}$$

where $v_{current}$ and v_{target} represent the current and target metric values, respectively. Based on the M/M/N queuing model, $T_{Analytical}$ is designed for high-concurrency scenarios where request buffering becomes significant. To balance computational efficiency with operational stability, we adopt a heavy-traffic approximation by assuming a saturated queuing state, i.e., $C(q_t, \rho) \to 1$. This approximation yields a conservative lower bound for the required instances, ensuring strict adherence to the SLA_{rt} even during volatile traffic fluctuations. The number of instances q_t is determined by:

$$q_t > \left\lceil \frac{\lambda_t}{\mu} + \frac{1}{\mu(RT^{upper} - \frac{1}{\mu})} \right\rceil \tag{7}$$

where the average response time RT_{avg} is modeled as the aggregate of queuing delay and service latency:

$$RT_{avg} = \frac{1}{q_t\mu - \lambda_t} + \frac{1}{\mu} \tag{8}$$

where μ represents the service capacity of a single container. T_{λ} handles scenarios outside the above categories by applying the steady-state equilibrium equation under ideal resource conditions, providing a baseline for stable QoS.

$$q_t = \lambda_t / \mu \tag{9}$$

2) Bound Protection Mechanism: To ensure the stability of environments, HTI incorporates a bound protection mechanism. Prior to the commitment of any scaling commands, HTI performs rigorous boundary validation on the parameters p_t generated by the LLM. For instance, the target CPU utilization is strictly constrained within the admissible interval $\mathcal{B} = [1, 100]$. The final executed parameter $\pi(p_t)$ is governed by the following logic:

$$\pi(p_t) = \begin{cases} p_t, & p_t \in \mathcal{B} \\ p_{safe}, & p_t \notin \mathcal{B} \end{cases} \tag{10}$$

where p_{safe} denotes a predefined default configuration. In the event of a constraint violation ($p_t \notin \mathcal{B}$), HTI automatically triggers this fallback mechanism, thereby ensuring the continuous and deterministic operation of the system. Table III is an example of LLM-generated reasoning chain.

TABLE III
AN EXAMPLE OF LLM-GENERATED REASONING CHAIN FOR AUTO-SCALING

Parameters	**Value**
Reasoning Steps	... It is known that the maximum request arrival rate of a single pod is 40 and the current number of pods is 5, so the maximum request arrival rate of the microservice at this time = 40 × 5 = 200. However, the request arrival rate of the microservice at this time is 253 and there are pending requests in the request queue, so it is determined that the request load is exceeded.
Selected Tool	Queuing analysis tool $T_{Analytical}$
Key parameter	Target single-pod maximum request arrival rate is 35 (adjusted from 40)
Final Action	Reduce single-pod maximum request arrival rate to 35 to alleviate queue congestion

C. System Feedback based Interactive Learning Method

To continuously optimize R-AM, SFIL utilizes a reward function r to quantify performance feedback and fine-tunes the LLM through Low-Rank Adaptation (LoRA), thereby promoting the strategy optimization of LLM. The reward design achieve a critical balance between SLA and resource provisioning costs. Formally, r is categorized into three operational scenarios based on the next-step state state of the microservice:

$$r_t = \begin{cases} -\rho_i \cdot q_t, & \text{if } RT_{95} > RT^{upper} \vee \lambda_t = 0 \\ \dfrac{RT_{95}}{\rho_i \cdot q_t}, & \text{otherwise} \end{cases} \tag{11}$$

when $RT_{95} > RT^{upper}$, the negative reward expands as response time and resource waste increases. Conversely, when no traffic is observed ($\lambda_t = 0$), the objective exclusively to minimizing resource overhead through a linear penalty. Finally, when the system operates stably, the reward is defined as the efficiency ratio of performance gain to leasing costs, guiding the LLM toward an optimal balance that minimizes q_t without compromising the SLA.

In contrast to conventional offline fine-tuning, SFIL introduces an experience filtering mechanism based on the reward function r, aimed at prioritizing reasoning chain with high SLA gains for policy evolution. This mechanism ensures that the LLM continuously converges toward high-efficiency operational strategies. At the end of each decision cycle d_c,

SFIL utilizes only the auto-scaling experiences that receive positive feedback (i.e., $r > 0$) into a tuple e_t, which is then archived in the experience replay buffer $\mathcal{D}$:

$$e_t = (H_t, rc_t) \tag{12}$$

where H_t represents the temporal semantic state and rc_t denotes the reasoning chain generated by the LLM. Consequently, the experience buffer can be formalized as $\mathcal{D} = \{e_k \mid r_k > 0\}_{k=t-N_\mathcal{D}}^{t}$.

When the sample size in the buffer reaches the threshold $N_\mathcal{D}$, the system employs LoRA for lightweight fine-tuning of the LLM. LoRA enables the LLM to rapidly capture the dynamic workload characteristics of specific applications while effectively preserving its inherent generalization capabilities. The optimization objective is defined as:

$$\min_{\Delta\Phi} \mathbb{E}_{e_t \in \mathcal{D}} \left[\mathcal{L}(f(H_t; \Phi + \Delta\Phi), rc_t) \right] \tag{13}$$

where Φ denotes the frozen parameters of the base model, and $\Delta\Phi = AB$ represents the weight increment composed of low-rank matrices A and B. By integrating LoRA, SFIL significantly enhances specialization of the LLM in OM tasks without compromising its foundational reasoning performance.

V. Performance Evaluation

Our approaches are compared to existing algorithms on a real Kubernetes based data center.

A. Baseline Methods

To comprehensively evaluate the performance of R-AM, we select five auto-scaling algorithms commonly used in cloud-native platforms as baselines for comparison:

- **HPA:** This scheme employs a threshold-based reactive triggering mechanism [6]. It monitors CPU or memory utilization in real-time and adjusts replica counts using a proportional scaling formula.
- **MMN:** This approach performs resource inference entirely based on the $M/M/N$ queuing theory model [9]. It predicts the minimum number of replicas required to satisfy SLA constraints based on statically preset service rate parameters.
- **SQF:** This approach represents a classic hybrid feedback control scheme [11]. SQF integrates static feedback control from state-space models with multi-layer coordinated control across queuing networks.
- **AR:** A mapping method based on static processing capacity [22]. This algorithm calculates the required scale by proportionally mapping the current workload against pre-measured instance capacity.
- **LLM-only:** This method directly uses LLMs as the decision-making core, taking qualitative microservice states as input to generate the number of containers for microservices [21].

B. Experimental settings

These experiments are conducted on a physical kubernetes cluster consisting of eleven machines, including one control plane, one monitor, one GPU server, and eight workers. The specific hardware configuration for each machine is detailed in Table IV. The monitor machines is equipped with Prometheus for high-frequency collection [23]. To support the real-time inference and fine-tuning requirements of R-AM, a GPU server with two GPUs is deployed to serve and fine-tune the LLM. Utilizing a distributed inference framework, this setup ensures that LLM response times remain within the operational decision cycles. The experimental benchmark employs the Ticket Sales (TS) application, a sophisticated cloud-native application. It comprises 42 heterogeneous microservices covering diverse business logics such as querying, booking, payment, and notification. Each microservice pod is constrained with resource limits of 0.5 CPU cores and 512MB RAM to simulate fine-grained resource scheduling. Furthermore, the leasing cost for each pod is set at \$0.000015/s, aligned with the pricing standards of Amazon EC2 t2.micro instances. The workload traces are derived from real-world Wikipedia access logs from September 19 to 25 in 2009, which represent typical load characteristics of traditional web applications. For the TS application, based on its characteristics such as the longest access path and network bandwidth, RT^{upper} for the business path is set to 1 seconds, V_{rt}^{sla} is 10%, and Δt is set to 1 minute. Experimental tests have shown that the execution time of LLM parallel inference, which ranges from 4.18s to 5.14s, can fully complete the task within the d_c.

TABLE IV
Hardware Configuration of the Kubernetes Cluster

Node Role	Specifications / Hardware
Control Plane	Kubernetes v1.27, Ubuntu Desktop 22.04, 15 vCPUs, 15GB RAM
Monitor	Ubuntu Server 22.04, 16 vCPUs, 16GB RAM
Worker	Ubuntu Server 22.04, 16 vCPUs, 16GB RAM
GPU Server	Ubuntu Desktop 24.04, NVIDIA RTX 5090, NVIDIA RTX 5090 D v2

C. Parameter Tuning

This section details the calibration of algorithmic parameters. The experimental evaluation encompasses the cumulative resource cost C_{cum} and the V_{rt} across multiple dimensions, including average response time RT_{avg}, median response time RT_{50}, and 95th percentile response time RT_{95}. HPA utilizes CPU utilization as the scaling metric, and its performance under different utilization thresholds is presented in Table V. Lower CPU utilization thresholds (e.g., 30% and 40%) provide an ample resource buffer for microservices, allowing them to better accommodate sudden traffic bursts. However, experimental results indicate that excessively low thresholds render the system hypersensitive to minor workload fluctuations. When small disturbances occur, these low thresholds trigger frequent expansion and contraction commands by the

HPA. The continuous creation and destruction of containers introduce substantial overhead. Such scheduling jitter consumes computational resources originally intended for processing business requests, leading to a significant deterioration in response times during scaling periods. Conversely, setting a higher CPU utilization threshold (e.g., 60%) can effectively reduce scaling frequency and maximize resource utilization. Nonetheless, when faced with sudden high-concurrency requests, the system often operates at the edge of saturation. This makes it highly susceptible to request backlogs and queuing before new pods become ready, resulting in severe SLA violations. Consequently, this paper sets the HPA CPU utilization threshold to 50%.

TABLE V
PERFORMANCE OF HPA UNDER DIFFERENT CPU UTILIZATION THRESHOLDS

CPU Utilization	V_{rt}			C_{cum}
	RT_{avg}	RT_{50}	RT_{95}	
30	30.17%	17.32%	54.19%	$4.63
40	18.99%	8.38%	39.66%	$3.38
50	11.17%	2.23%	26.82%	$2.03
60	89.39%	0.0%	100.0%	$1.56

To ensure reproducibility and operational stability, the deployment of the LLM within the R-AM framework has been standardized. The core parameters of the LLM are summarized in Table VI. In this paper, model fine-tuning and exportation are conducted using the LlamaFactory framework, with vLLM serving as the high-performance inference backend. We select Qwen-14B as the base model, as it exhibits significantly stronger logical reasoning capabilities compared to 7B-scale models [24]. Simultaneously, compared to ultra-large-scale 70B models, the 14B model offers lower inference latency in a dual-GPU parallel environment, meeting the real-time requirements of large-scale microservice clusters. To minimize decision errors caused by numerical fluctuation, R-AM employs full-precision inference. The system is configured with a Tensor Parallel size of 2, distributing the computational load across GPUs via tensor-parallel computing. Furthermore, to mitigate the risk of Out-of-Memory errors triggered by fluctuating context lengths under dynamic workloads, the GPU memory utilization is strictly capped at 80%. The Max Model Len is set to 4,096 tokens, which is sufficient to cover all input sequences. In concurrent multi-service scenarios, vLLM efficiently handles inference requests with Max Num Seqs limited to 16. This configuration balances throughput and queuing latency, preventing the degradation of decision timeliness caused by excessive concurrency. Finally, we employ LoRA as the fine-tuning strategy.

D. Ablation experiments

This section details the implementation of ablation studies. we constructed two ablation variants for comparative experiments, and the experimental data are summarized in Table VII.

TABLE VI
LLM INFERENCE AND TRAINING HYPERPARAMETERS

Category	Parameter	Value
Architecture	Framework	LlamaFactory
	Base Model	Qwen-14B
	Quantization	None
vLLM Config	Tensor Parallel	2
	GPU Utilization	0.8
	Max Model Len	4,096
Inference	Infer Backend	vLLM
	Enforce Eager	Enabled
	Max Num Seqs	16
Fine-tuning	Tuning Method	LoRA
	Template	Qwen

1) The Necessity of RAG: In this variant, the RAG was removed, forcing the LLM to generate reasoning chains directly based on the semantic descriptions. As shown in Table VII, without RAG $O\text{-}RAG$, the V_{rt} of RT_{95} increased by 4.47% (from 6.70% to 11.17%), while cumulative costs rose by 5.62% (from $6.76 to $7.14). This performance gap arises because, without the guidance of expert heuristics provided by RAG, the LLM relies solely on raw numerical data. This lack of contextual grounding leads to the generation of erroneous reasoning chains, as the model struggles to bridge the semantic gap between metric fluctuations and systemic root causes. The results show that RAG can effectively improve the logical reasoning accuracy of large language models in complex scenarios.

2) The Necessity of Fine-tuning: This variant disabled the LoRA incremental updates of the SFIL. As shown in Table VII, the lack of this continuous learning mechanism resulted in significant degradation of system stability. The V_{rt} of RT_{95} surged by 11.74% (reaching 18.44%), while V_{rt} of RT_{avg} and RT_{50} represented the worst performance among all configurations. Meanwhile, costs increased by 22.18%. These results provide compelling evidence for the decisive role of continuous learning mechanism in dynamic cloud environments. Without the SFIL $O\text{-}SFIL$, the R-AM cannot rectify its strategies based on actual execution feedback, leading to a severe decoupling between decision logic and real-time environment states.

As shown in Table VII, R-AM achieved the lowest V_{rt} of RT_{95} (6.70%) while maintaining the minimum operational cost ($6.76). In contrast, the absence of any individual component led to either escalated costs or degraded SLA.

TABLE VII
RESULTS OF ABLATION ANALYSIS FOR R-AM

Algorithms	V_{rt}			C_{cum}
	RT_{avg}	RT_{50}	RT_{95}	
R-AM	1.68%	1.12%	6.70%	$6.76
O-RAG	0.56%	0.00%	11.17%	$7.14
O-SFIL	4.47%	2.13%	18.44%	$8.26

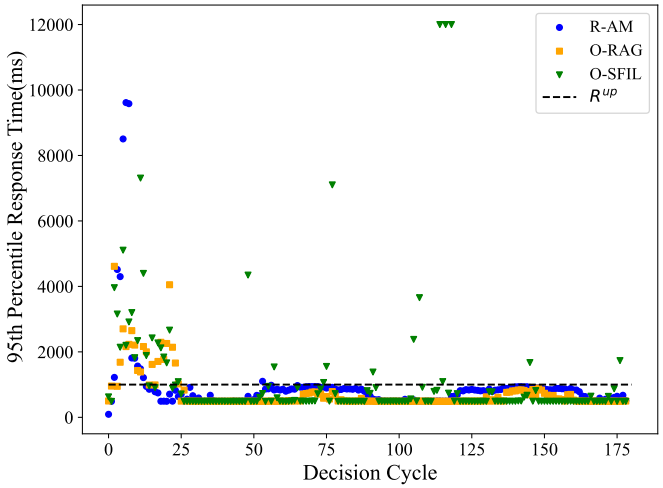

Fig. 5. Comparison of RT_{95} for R-AM and its variants

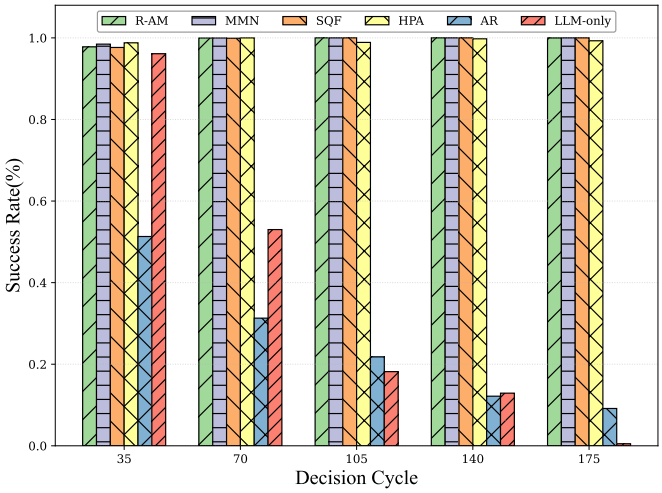

Fig. 6. Trend of success rate over decision cycles

E. Comparison Results

Several widely metrics is used to evaluate the performance of our proposed method R-AM and other existing algorithms, including the success rate, the violation of response time and the cost. The results are shown in Table VIII.

TABLE VIII
RESULTS OF R-AM AND OTHER ALGORITHMS

Algorithms	Success Rate	V_{rt}			C_{cum}
		RT_{avg}	RT_{50}	RT_{95}	
R-AM	99.55%	1.68%	1.12%	6.7%	$6.76
MMN	99.69%	2.79%	1.68%	11.17%	$7.13
SQF	99.52%	8.94%	1.68%	12.29%	$3.05
HPA	99.35%	11.17%	2.23%	26.82%	$2.03
AR	25.64%	99.44%	6.7%	99.44%	$0.96
LLM-only	35.33%	97.77%	59.22%	98.32%	$6.21

1) Comparison of Success Rate: Success rate measures the proportion of requests completed within the specified timeout period. As shown in Table VIII, R-AM achieves a success rate of 99.55%, which is on the same order of magnitude as MMN (99.69%) and superior to HPA (99.35%). This performance demonstrates the inherent stability of R-AM during the scaling process. In contrast, AR (25.64%) and LLM-only (35.33%) exhibit exceptionally high failure rates. As shown in Fig. 6, both methods initially maintained success rates between 51% and 96% during the early stages of the experiment (35th decision cycle). However, as the decision cycles progressed, both experienced significant performance degradation. By the end of the experiment (175th cycle), the success rate of LLM-only converged to zero, while AR plummeted to below 1%. The primary reason for this failure is the significant cascading effects among microservices when facing complex burst workloads. Both AR and the LLM-only approach fail to accurately locate bottleneck microservices, leading to the misallocation of resources across the microservice chain.

2) Comparison of Response Time: The Cumulative Distribution Function (CDF) of response times illustrates the degree of latency control maintained by each algorithm. Beyond the 1,000ms threshold, the CDF curve of R-AM exhibits the steepest ascent, achieving the most rapid convergence to 1.0. As shown in Table VIII, the V_{rt} for R-AM at the RT_{95} is only 6.70%, representing 20.12% reduction compared to HPA. Within the critical workload interval of 500ms to 1,000ms, R-AM maintains robust CDF growth. The V_{rt} for R-AM at the RT_{50} stands at a 1.12%, significantly outperforming other algorithms. This reflects the robustness of its decision mechanism under normal fluctuations. In contrast, the CDF curves for AR and LLM-only remain nearly horizontal before the 1,000ms, indicating that the vast majority of requests experience latencies exceeding 10,00ms, with V_{rt} of RT_{95} approaching 100%. This issue originates from a misalignment between state characterization and policy execution, where AR and LLM-only exhibit poor robustness against abnormal state fluctuation. Experimental results show that R-AM achieves the lowest V_{rt} among all algorithms.

3) Comparison of Resource Cost: The evaluation of resource costs reveals the capability of algorithms to balance QoS with resource utilization efficiency. As shown in Table VIII, the C_{cum} of R-AM is $6.76, which is the lowest among all algorithms that maintain a high success rate and a low V_{rt}. Compared to MMN ($7.13), R-AM achieves a 5.19% optimization while providing superior QoS. This indicates that through the guidance of expert knowledge and the continuous learning of LLM, R-AM effectively mitigates the over-provisioning issues typically associated with traditional models. While the C_{cum} of HPA ($2.03) and SQF ($3.05) are lower, they come at the expense of the V_{rt}. Such low expenditures essentially stem from systemic instability caused by severe under-provisioning. The LLM-only baseline yields a severe V_{rt} of 98.32% at a cost of $6.21. This performance degradation is primarily attributed to the precision loss in-

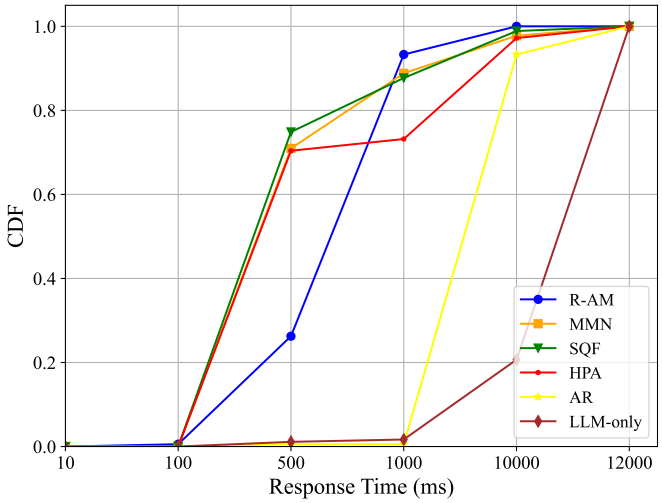

Fig. 7. CDF of 95th percentile response time

curred during the transformation from numerical metrics to semantic representations. The results indicate that pure semantic representations (i.e., representations that discard raw numerical data) are insufficient to accurately describe microservice state. Therefore, the reliance of LLMs on the semantic representations can lead to imprecise decisions regarding container replicas. Furthermore, the research result also show that in the absence of scheduling logic, the resources allocated by LLM is not able to effectively improve service quality.

VI. Conclusions and Future Work

This paper proposes a RAG-driven Auto-scaling Method (R-AM) to ensure QoS. The proposed RAG-based Decision Reasoning (RDR) method is able to enhance the interpretability and robustness of models. By integrating metric sensitivity analysis, scaling tool recommendation, and parameter generation, this method achieves a decoupling between decision logic and the numerical distribution of monitoring metrics. Based on the reasoning chain, the LLM invokes specially tools from a heterogeneous library to execute auto-scaling action. Through continuous interaction with the kubernetes cluster, R-AM achieves iterative policy refinement based on system feedback, effectively eliminating the reliance on manual fine-tuning typical of traditional LLM applications. Experiments show that R-AM reduces the V_{rt} of RT_{95} from 11.17% to 6.7% while enhancing the success rate by 73.91%. Exploration of Large Language Model Operations (LLMOps) for cloud-native applications is a promising research direction.

References

[1] J. Kosińska and K. Zieliński, "Experimental evaluation of rule-based autonomic computing management framework for cloud-native applications," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1172–1183, 2023.

[2] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 13–18.

[3] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[4] S. Zhang, C. Wang, and A. Y. Zomaya, "Robustness analysis and enhancement of deep reinforcement learning-based schedulers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 346–357, 2023.

[5] Z. Cheng, J. Yu, and X. Xing, "A survey on explainable deep reinforcement learning," *arXiv preprint arXiv:2502.06869*, 2025.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, Apr. 2016.

[7] S. Merkouche and C. Bouanaka, "A hybrid approach for containerized microservices auto-scaling," in *2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2022, pp. 1–6.

[8] D. Trihinas, Z. Georgiou, G. Pallis, and M. D. Dikaiakos, "Improving rule-based elasticity control by adapting the sensitivity of the auto-scaling decision timeframe," in *Algorithmic Aspects of Cloud Computing*, D. Alistarh, A. Delis, and G. Pallis, Eds. Cham: Springer International Publishing, 2018, pp. 123–137.

[9] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1994–2004.

[10] Z. Cai and R. Buyya, "Inverse queuing model-based feedback control for elastic container provisioning of web systems in kubernetes," *IEEE Trans. Comput.*, vol. 71, no. 2, p. 337–348, Feb. 2022.

[11] Y. Lei, Z. Cai, X. Li, and R. Buyya, "State space model and queuing network based cloud resource provisioning for meshed web systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3787–3799, 2022.

[12] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "esdnn: Deep neural network based multivariate workload prediction in cloud computing environments," *ACM Trans. Internet Technol.*, vol. 22, no. 3, Aug. 2022.

[13] F. Zhao, W. Lin, S. Lin, S. Tang, and K. Li, "Mscnet: Multi-scale network with convolutions for long-term cloud workload prediction," *IEEE Transactions on Services Computing*, vol. 18, pp. 969–982, 2025.

[14] D. Saxena, J. Kumar, A. K. Singh, and S. Schmid, "Performance analysis of machine learning centered workload prediction models for cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1313–1330, 2023.

[15] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 50–56.

[16] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: an intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[17] H. Bai, M. Xu, K. Ye, R. Buyya, and C. Xu, "Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 3473–3484, 2024.

[18] W. Zou, Z. Zhang, N. Wang, Y. Tian, and L. Tian, "Reflexpilot: Startup-aware dependent task scheduling based on deep reinforcement learning for edge-cloud collaborative computing," *IEEE Transactions on Cloud Computing*, vol. 13, no. 2, pp. 641–654, 2025.

[19] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Unilog: Automatic logging via llm and in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[20] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, ser. CIKM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 4966–4974.

[21] R. Chen, Y. Pu, J. Xin, J. Wang, X. Liao, K. Zhang, and W. Wu, "Grace: A strategic llm-enhanced graph reinforcement learning framework for

adaptive fault recovery in microservice systems," in *Service-Oriented Computing: 23rd International Conference, ICSOC 2025, Shenzhen, China, December 1–4, 2025, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2026, p. 155–170.

[22] F. Obaid and S. Ahmed, "Feedback control of computing systems," in *2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, 2019, pp. 1–5.

[23] N. Sukhija and E. Bautista, "Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus," in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2019, pp. 257–262.

[24] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.