



RESEARCH ARTICLE

OPEN ACCESS

ENERGY AND QOS-AWARE MICROSERVICE RESOURCE MANAGEMENT USING MULTI-OBJECTIVE MULTI-TASK REINFORCEMENT

Sannam Nagendram¹, Medeswararao Kondamudi², Rajesh Yamparala³
Tirumalasetti Lakshmi Narayana⁴, Sunitha Pachala⁵, Shobana Gorintla⁶, Koppolu Sireesha⁷
Inakoti Ramesh Raja⁸, Garaga srilakshmi⁹, Desamala Prabhakara Rao¹⁰

¹Professor, Department of CSE (AIML), KKR & KSR Institute of Technology and Sciences, Guntur, India.

^{2,3}Computer Science & Engineering (Data Science), R.V.R. & J.C. College of Engineering, Chandramoulipuram, Chowdavaram, Andhra Pradesh - 522 019, India.

⁴Assistant Professor, Department of Electrical and Electronics Engineering, Aditya university, Surampalem, A.P, India

⁵Associate Professor, Computer Science and Engineering, Koneru Lakshmaiah Education Foundation, Green Fields, Vaddeswaram, India.

⁶Professor, Department of CSE, Department of CSE, NRI Institute of Technology (Autonomous), Pothavarapadu, Agiripalli Mandal, Pincode- 521212, Eluru District, Andhra Pradesh, India.

⁷Assistant professor, Department of CSE-AIML, R. V. R. & J.C College of Engineering, Guntur, Andhra Pradesh, India.

^{8,9}Associate Professor, Department of Electronics and Communication Engineering, Aditya University, Surampalem, A.P., India.

¹⁰Associate Professor, Department of Electronics and Communication Engineering, Chalapathi Institute of Technology, Mothadaka, Guntur, India.

¹<http://orcid.org/0000-0002-7265-1860>, ²<http://orcid.org/0009-0002-9911-5731>, ³<http://orcid.org/0000-0003-1412-7240>

⁴<http://orcid.org/0009-0000-5359-6723>, ⁵<http://orcid.org/0000-0002-4149-7178>, ⁶<https://orcid.org/0000-0002-9357-6142>

⁷<http://orcid.org/0009-0000-7620-9001>, ⁸<https://orcid.org/0009-0006-6806-2605>, ⁹<https://orcid.org/0009-0005-5254-7766>

¹⁰<https://orcid.org/0009-0001-3930-3258>

E-mail: *reena1286@gmail.com, kmedeswararao@gmail.com, rajeshyamparala@gmail.com
tlaxman17@gmail.com, drsunitha.pachala@kluniversity.in, drgshobana@gmail.com, koppelusireesha@rvrjc.ac.in
inakoti.rameshraj@gmail.com, srilakshmi1853@gmail.com, prabhakardesamala@gmail.com

ARTICLE INFO

Article History

Received on: January 21, 2026

Reviewed on: February 21, 2026

Accepted on: April 10, 2026

Published on: June 30, 2026

Keywords:

Eeg.

Frame Work.

Lstm.

Cnn.

ABSTRACT

The paper talks about microservice architecture. This kind of design splits a big application into small services. Each service works on its own. These services are able to be placed on different servers. This makes systems more flexible and scalable. Most past works only focus on reducing delay. But delay is not the only concern. Power use is important. High power use leads to high cost and poor sustainability. So, this paper looks at both delay and power together. A new system termed GIRP was proposed. GIRP stands for Green and Intelligent Resource Provisioning. It tries to manage both delay and power. It does this by controlling CPU usage and service copies. These are placed on servers smartly. To make good decisions, GIRP uses a special learning method. This method is titled as Deep Reinforcement Learning. It learns by trying different actions. It sees what works best. Then it learns from that. In this work, a multi-Task learning model is used. It decides two things at the same time: the number of CPU cores to use and the number of service replicas to create. This helps in managing resources well. Another part of the system handles placement for each service. GIRP is tested using a popular benchmark. The results are strong. This proves GIRP works well. This paper shows a smart way to manage microservices. It saves power and meets user delay needs. The learning model makes better choices over time.



Copyright ©2026 by authors and Galileo Institute of Technology and Education of the Amazon (ITEGAM). This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

I. INTRODUCTION

Microservice architecture has become very popular in recent years [1]. It helps developers create and manage large applications. In this method, each part of the system is built as a small and separate service. These services work independently. Each microservice is responsible for a single task or feature [2]. These services communicate using standard protocols like RPC (Remote Procedure Call). This makes the system more flexible. It improves the scalability and performance of applications. One key advantage of microservices is the ability to scale easily [3]. When there is a high number of user requests, only the needed services are scaled. This is better than scaling the whole application as in monolithic systems. It saves resources and makes updates faster. Microservices are deployed in containers or virtual machines [4]. Each service has its own CPU, memory and other resource limits. Microservices bring new challenges. Managing so many small services is not easy [5]. It needs smart techniques to handle resource allocation and service scheduling. These decisions affect both performance and power usage. Many previous works focused only on reducing delay or latency. The purpose is to make sure the application responds quickly to users, but power consumption is usually ignored. Using too many servers or resources waste energy. Today, power efficiency is very important [6]. Data centers consume a lot of energy. Reducing energy use helps the environment and saves money. So, both delay and energy use should be considered together. This paper focuses on both objectives.

It wants to reduce end-to-end delay and power consumption together. This is not easy. There are many challenges in this system. Delay and power work against each other [7]. Reducing delay uses more power. Saving power causes more delay. Decisions are discrete and hard to optimize. The effect of each action is unclear. Modeling the system is not easy [8]. GIRP system uses data to make smart decisions. It adjusts resource allocation and service placement in real time. GIRP tries to find a good balance between delay and power [9]. It uses deep reinforcement learning (DRL) to learn the best actions. DRL allows the system to improve over time. It does not need a perfect model of the system. In GIRP, there are two parts to decision making [10]. One part selects the number of CPU cores and service replicas to use. The other part decides the placement of each service on the servers. A multi-task learning model uses a structure named Multi-gate Mixture-of-Experts (MMoE). This helps it learn two tasks at the same time. It decides both the CPU and replica numbers together [11]. For the second part, another learning agent is used. It learns the best way to distribute services across servers. The system is tested using a named benchmark, DeathStarBench. It includes two applications: Hotel Reservation and Social Network. These are made of many microservices. These are good test cases for evaluating real systems. The key contributions of this paper are:

- Develop a system that addresses both latency reduction and energy efficiency, unlike existing methods that focus on only one aspect.
- Design the GIRP framework as a real-time end-to-end solution, including components for data collection, performance analysis, intelligent decision-making and automated deployment, with the ability to adapt to changing workloads.
- Implement a multi-task learning strategy using a Multi-gate Mixture-of-Experts (MMoE) model to optimize CPU and replica allocation simultaneously, reducing training cost and increasing accuracy.
- Apply Deep Reinforcement Learning (DRL) to drive system actions, allowing continuous improvement through learning and exploration without relying on fixed rules, validated on real microservice workloads.
- Confirm scalability of the GIRP architecture to support applications of various sizes and complexity, across different server configurations and microservice setups.
- Enable balanced decision-making by evaluating multiple factors for example resource load, user demand and energy constraints to maintain efficient and reliable service delivery.

Microservices are very useful [12]. These test cases are hard to manage well. Systems are needed that adjust resources smartly. These systems should balance delay and power use [13]. This paper presents GIRP. It is a new system that solves these challenges using deep learning. It learns from data. It makes good decisions about resources and service placement [14]. It saves energy and keeps users happy. The key approach is to treat resource control as a learning problem. With the help of reinforcement learning and multi-task learning, GIRP learns to manage resources better over time [15]. This helps modern systems become more efficient and sustainable.

II. RELATED WORK

These past research works focus on microservice scheduling and resource allocation. Many researchers studied delay and energy use in such systems. But most of them focused on only one part — either reducing delay or saving energy. Some studies used cloud platforms like AWS. For example, in [16] a tool was created that adjusts CPU cores and service replicas. It uses system usage to scale resources. This helps reduce delay and avoids using too much hardware. In another work, Sinan [17] used a multi-armed bandit method. This helps in choosing resource levels based on changing user requests. It works well under dynamic loads. It does not look at service deployment strategies or full system energy. Firm [18] used reinforcement learning. It tries to reduce delay caused by CPU, memory and network problems. It changes CPU cores during run-time. It increases or decreases service replicas based on core needs. But it does not plan for server-level placement. Other approaches used single-task learning models. Each model handles one task, like deciding CPU or replicas [19]. This leads to extra work and training time. These models cannot share knowledge between tasks. So, such approaches are not suitable for complex systems. Traditional systems used rule-based or static policies. These methods cannot handle changing traffic patterns. The approaches are simple but lack intelligence and fail under dynamic user loads [20], [21]. In recent years, many researchers suggested deep reinforcement learning (DRL) methods. These methods learn from system data. For example, in [22] DRL is used to reduce delay by making smart resource choices. Other studies tried to save energy in data centers. Green computing is now a key concern. Some works reduce the number of active servers. Others try to balance the load better [23], [24]. Few studies attempted to address both objectives at the same time. A few of them use optimization frameworks. But these need accurate system models.

The GIRP framework tries to solve these problems together. It uses multi-task learning and DRL. It manages CPU, replicas and service placement. It learns from past data. It adjusts to user load and resource usage. It saves power and meets delay needs. A model named MMoE was used [25]. It is good at handling many tasks together. It allows sharing of knowledge. This makes training faster and more accurate. It avoids repeating the same work. Some works used MMoE in other fields. For example, [26] applied it in advertising. It helped to improve predictions in different user groups. But GIRP is the first to use it for microservice resource control. Most past studies focus on only part of the problem. These approaches do not manage CPU, replicas and placement together. In many cases, power is ignored. The GIRP framework tries to fill this gap.

III. SYSTEM MODELLING & PROBLEM FORMULATION

This section explains the way the GIRP framework models the system and defines the problem. The purpose is to make microservices run efficiently by managing resources wisely. GIRP tries to reduce both the delay experienced by users and the power consumed by servers. To do this, it creates a mathematical model of the system. First, the model considers a fixed service period. This period is named as T_0 and is divided into equal time intervals. Each time interval is denoted as δ . During each time step s , two things are measured: the end-to-end service latency q_s and the total power usage E_s . The latency is recorded in milliseconds, while the power is measured in joules. The model defines a target latency value Q_0 . If the observed latency q_s goes above Q_0 , the quality of service (QoS) is violated. There are M microservices and N servers in the system. Each microservice is placed on exactly one server. This placement is modeled using a binary variable $z_{i,j}$. If service i is placed on server j , then $z_{i,j} = 1$; otherwise, it is 0. This constraint is written as:

$$\sum_{j \in N} z_{i,j} = 1, \quad i \in M \quad (1)$$

Each microservice gets some CPU cores and replicated multiple times. If a service is placed on a server, then the system decides the number of CPU cores $x_{i,j}$ to assign and the number of replicas $y_{i,j}$ to create. The delay for running service i on server j is represented by a function $f(x_{i,j}, y_{i,j})$. This function captures the effect of CPU and replicas on latency. The total latency of the system is calculated by summing the delays from all services on all servers. The equation for total latency q is:

$$q = \sum_{j \in N} \sum_{i \in M} z_{i,j} f(x_{i,j}, y_{i,j}) \quad (2)$$

Power consumption is calculated in a similar way. The function $g(x_{i,j}, y_{i,j})$ represents the CPU usage of service i on server j . Using this function, compute the total CPU load μ_j^{CPU} on server j as:

$$\mu_j^{\text{CPU}} = \sum_{i \in M} z_{i,j} g(x_{i,j}, y_{i,j}) \quad (3)$$

Then, use μ_j^{CPU} to find the power usage E_j for each server. This is calculated as:

$$E_j = (\mu_j^{\text{CPU}} + \xi) P_j^R \delta \quad (4)$$

In this formula, P_j^R is the rated power of server j and ξ is a constant that represents the operating system's power usage. The time interval δ is the same as defined earlier. The purpose of GIRP is to reduce both latency and power use. To do this, it makes decisions about three things: CPU allocation x , replica allocation y and service placement z . These are grouped into three decision matrices: $x = \{x_{i,j}\}$, $y = \{y_{i,j}\}$ and $z = \{z_{i,j}\}$. The main problem is written as a multi-objective optimization. The system chooses x , y and z to minimize both total latency and total power consumption. This problem is formally defined as:

$$(P1) \quad \min_{x,y,z} \left\{ \sum_{j \in N} E_j, \quad q \right\} \quad (5)$$

This optimization is subject to several constraints. First, CPU and replica allocations are limited to a certain maximum value. CPU cores $x_{i,j}$ are set as integers between 1 and $C_{\max}$. Replicas $y_{i,j}$ are chosen between 1 and $R_{\max}$. The placement $z_{i,j}$ is binary (0 or 1) and each service is placed on one and only one server. These conditions are written as:

$$x_{i,j} \in \{1, 2, 3, \dots, C_{\max}\}, \forall i, j \quad (6)$$

$$y_{i,j} \in \{1, 2, \dots, R_{\max}\}, \quad \forall i, j \quad (7)$$

$$z_{i,j} \in \{0, 1\}, \quad \forall i, j \quad (8)$$

$$\sum_{j \in N} z_{i,j} = 1, \quad \forall i \quad (9)$$

This is a difficult problem to solve. All decision variables are discrete. The exact forms of the functions f and g are not available. Another difficulty is that power and latency opposes each other. Reducing one makes the other worse. For example, adding more replicas lowers latency but increase power consumption. Traditional rule-based systems are not good at handling these trade-offs. Such approaches are usually static and cannot respond well to changing workloads. Therefore, this paper uses DRL. DRL learn from experience. It does not need an exact model of the system. Instead, it interacts with the environment and improves over time. GIRP divides the problem into two parts. The first part is about resource allocation, which includes vertical scaling and horizontal scaling (replicas). The second part is about service placement. To solve these parts, GIRP uses two separate learning agents. The first agent is titled as RA agent. It handles CPU and replica allocation. The RA agent uses a learning structure termed MMoE.

This model allows the system to learn two tasks at the same time. It has shared layers to learn common knowledge and expert layers for task-specific knowledge. This setup helps reduce the training time and increases performance. The second agent is named as the SA agent. The SA agent focuses on a single task and learns placement strategies that optimize both power and latency. By using these two agents, GIRP handles the whole problem in a smart and dynamic way. The system keeps learning from data. It adjusts its resource usage and service placement based on real-time feedback. This modeling approach is very useful. It allows the system to understand the impact of its actions on both delay and energy. This helps it make better decisions that balance user needs with energy efficiency. This section presents a formal model for resource provisioning. It includes key equations for latency and power. It outlines the system's constraints. Finally, it describes the way the GIRP framework solves the problem through multi-agent deep reinforcement learning. The combined modeling and learning approach helps GIRP deliver good quality of service while using less energy.

IV. SYSTEM DESIGN OF GIRP FRAMEWORK

GIRP is designed to manage microservices in a smart and efficient way. The purpose is to reduce power usage while keeping delays low. This is important for real-world systems with changing workloads. The design of GIRP is based on four main principles. The first focus is latency minimization. GIRP is designed to reduce the end-to-end delay in serving user requests. It targets meeting the 99th percentile latency under all system loads. The second focus is energy efficiency, using less energy while still maintaining the quality of service. The third focus is integration, linking vertical scaling, horizontal scaling, and service scheduling. These components influence one another. The final focus is adaptiveness. The system adapts to load changes and user demands in real time. The GIRP framework has five major components. These include a real-time data collector. An online service profiler is used. There is an energy-latency-aware resource allocation and scheduling manager. A resource coordinator is included. The system has a deployment module. These components work together during each service cycle. Each service cycle is termed as T_0 . During T_0 , the framework checks if the current resource settings meet the latency objective. If everything is fine it continues with the same setup.

But if no problems are seen for a few cycles, it will still update to improve energy savings. The data collector gathers resource usage data from all running services. It checks CPU usage, memory use, network stats and disk reads/writes. Each server has a local probe. These probes use Docker tools to collect live data from each microservice container. The collector talks to an API gateway. It collects the number of active users, requests per second and 99th percentile latency. The RA agent handles CPU cores and replicas. The SA agent decides which server runs each microservice. These agents use deep reinforcement learning (RL) methods. The purpose is to reduce both delay and energy. The RA agent works on two tasks at once. It decides the number of CPU cores and service replicas. It uses a special model termed MMoE. This model helps learn both tasks together. Shared layers handle general patterns. Expert layers handle task-specific parts. This way, RA makes better and faster decisions. The SA agent focuses on service placement. It looks at the current system load and server status. Then it chooses the best server for each microservice. It uses RL. Its purpose is to balance delay and power use. Both RA and SA produce action probabilities.

These are numbers between 0 and 1. The outputs show the amount of each resource to use and the placement of a service. The next step involves the resource coordinator. This unit takes the action probabilities and turns them into real numbers. Finally, the deployment module makes the changes. It looks at the current setup and compares it with the new decisions. If something has changed, it updates the affected microservices. This is done using Docker's native command-line interface (CLI). The APIs let us set limits on CPU, memory, replicas and service labels. The flow of data and decisions in GIRP helps the system stays responsive. If latency goes above the threshold GIRP reacts fast. It gathers new data, updates its models and applies new actions. If no latency problem is found for a while, GIRP still runs a background check. It looks for ways to save more energy without hurting service quality. This makes the system proactive. The GIRP system is made of multiple connected parts. The use of deep learning and adaptive control helps the system deal with load changes and save power. By regularly collecting data, making smart decisions and applying changes on the fly GIRP becomes a strong framework for modern cloud systems. It keeps delay low and power use under control.

V. PROPOSED MULTI-TASK RL ALGORITHM TO (P1)

This section explains the proposed multi-task RL method that solves the optimization problem defined earlier as P1. To achieve this, two learning agents are designed. The first one is the RA agent, which handles the task of resource allocation. The second one is the SA agent, which decides the placement of each microservice.

These two agents work together inside the GIRP framework. The Figure 1 shows the complete structure of the GIRP control system that uses Multi-Agent Deep Reinforcement Learning. It starts with the environment that includes hardware resources like CPU, memory, disk, network, cache and LLC. Performance metrics for example the number of users, RPS (requests per second) and latency are measured. These values form the state input, which is marked as s_t . This state is passed into the actor network. The actor network is based on the MMoE model. It has shared expert layers (Expert 0, 1, 2) and task-specific gates. Each gate decides the portion to use from each expert. Then, separate output towers generate two actions: Action A and Action B. These actions are used to control CPU core allocation and replica count. The selected actions are passed to the environment and stored in an experience buffer, along with the state and reward r_t .

These samples are used for training. A critic network evaluates the quality of the action. The online critic network gives feedback, while a separate target critic network provides stable targets. Both actor and critic networks include target versions that are slowly updated using soft updates. The actor network uses the updated policy to select better actions in the future. The setup uses random noise to explore different actions during learning. This framework allows GIRP to balance delay and energy by making smart and adaptive resource decisions. RL is used to solve problems involving a sequence of decisions made by an agent over time. These decisions are made by observing the environment and receiving feedback. This setup is termed as Markov Decision Process (MDP). In every time step t , the agent sees a state s_t , picks an action a_t and receives a reward r_t . The purpose of the RL agent is to learn a policy $\pi_\theta(s)$, which tells the agent what to do in each state. The agent wants to maximize the expected total reward over time, as shown in the formula:

$$\max_{\pi_\theta} L_{\pi_\theta}(s, a) = E_{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s \right] \quad (10)$$

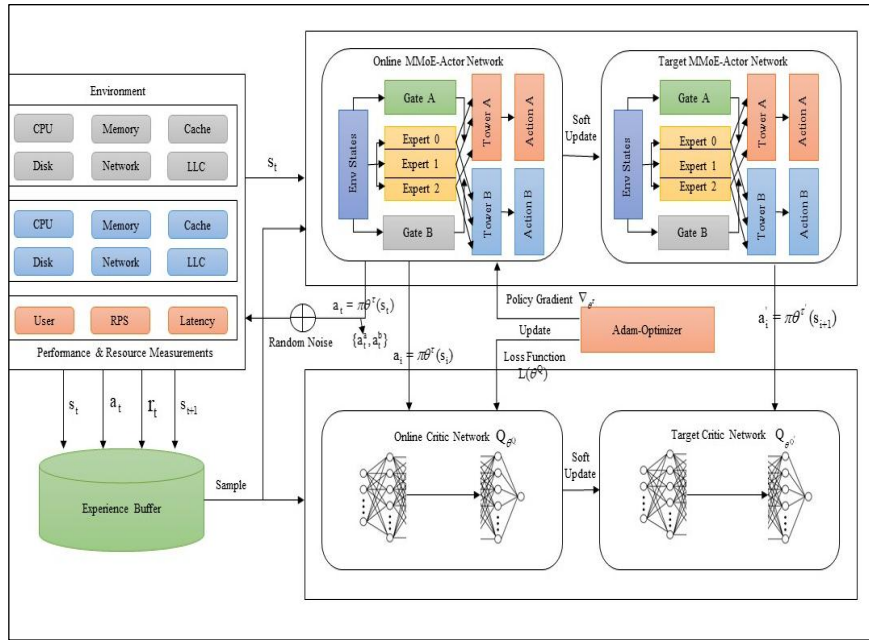


Figure 1: The Training Process of a MMoE-base Multi-task RL agent.
Source: Authors, (2026).

Here, γ is a discount factor between 0 and 1. It gives more weight to recent rewards and less to future ones. The algorithm used in GIRP is based on a method Deep Deterministic Policy Gradient (DDPG). This method uses two neural networks: one is the actor, which suggests the actions and the other is the critic which evaluates those actions. These networks are updated by using the observed rewards. The MMoE model works by having a group of expert networks. Each expert network is a small neural network. A gating network decides the contribution of each expert to the final decision. The output of the model is a weighted sum of expert outputs:

$$f_k(x) = \sum_{i=1}^n g_k(x) f_i(x) \quad (11)$$

Here, $f_i(x)$ is the output of the i -th expert. $g_k(x)_i$ is the weight given by the gating network for task k . These weights sum to 1 and are calculated using a softmax function:

$$g_k(x) = \text{softmax}(W_{g_k}, x) \quad (12)$$

The output for task k is then passed to a tower network h_k that gives the final decision:

$$y_k = h_k(f_k(x)) \quad (13)$$

This setup lets both tasks (CPU and replicas) share some information while learning task-specific behavior. This improves both speed and accuracy of training. The DDPG method is used to train both the actor and critic networks. The actor suggests an action based on the current state. The critic evaluates the quality of the action. The critic's loss function is defined as:

$$L(\theta_Q) = (Q(S_t, a_t \parallel \theta_Q) - y_t)^2 \quad (14)$$

The target value y_t is computed as:

$$y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}/\theta_\tau) / \theta_Q) \quad (15)$$

The critic is trained to minimize this loss. Then the actor is updated using the following gradient:

$$\nabla_{\theta_\tau} = \mu * \nabla Q\left(S_t, \pi\left(\frac{S_t}{\theta_Q}\right)\right) * \nabla \pi(S_t) \quad (16)$$

These steps are repeated at every training cycle. The target networks are updated slowly using a soft update rule to stabilize training. The RA agent produces two actions: a_t^a and a_t^b for CPU and replica scaling respectively. These are values between 0 and 1, which are then converted into actual resource values by multiplying them with the maximum available resources. The reward function for the RA agent is defined as:

$$r_t = \alpha * \frac{Q_0 - q_t}{c - \arctan(q_t - Q_0)} + (1 - \alpha) * \exp\left(\frac{1}{E_t - E_R}\right) \quad (17)$$

Here, q_t is the observed latency, E_t is the power consumed. Q_0 is the latency target and E_R is the reference power level. The parameter α determines the weight given to latency compared to power. The SA agent is trained using a similar reward function but has only one action: choosing the server for deployment. It outputs a probability distribution over the servers. The final server is selected based on the highest probability. This multi-task approach has many benefits. First, it allows both tasks to learn together and share useful knowledge. Second, it reduces the number of parameters compared to training two separate models. Third, it improves generalization and prevents overfitting. Lastly, it makes the system faster in decision-making during real-time operation. Both RA and SA agents use neural networks built with PyTorch. These networks include expert modules, tower layers and critic modules. Dropout is used to avoid overfitting. All parameters are tuned carefully to support high performance in real deployments. The RA and SA agents each use deep neural networks for decision-making. These networks are implemented using PyTorch. Special components are included to improve learning speed and accuracy. The RA agent uses the MMoE model, while the SA agent uses a standard DDPG structure. For the RA agent, the input is the current system state S_t . This includes values like user load, server utilization, CPU stats and memory data.

These features are passed through a set of shared layers. These layers extract general knowledge useful for both scaling tasks. After that, the information is sent to multiple expert networks. Each expert is a small fully-connected network. These experts produce different perspectives on the input. Then, a task-specific gate decides which experts to trust more. This is done separately for CPU scaling and replica scaling. Each gate produces a softmax weight vector. The weighted sum of expert outputs forms the input to a task-specific tower. The tower is another neural layer that gives the final action output. This output is a number between 0 and 1. It is later scaled to the real-world range. The critic network in RA has a simpler structure. It takes the state and action as input and outputs a Q-value. The critic is trained using the Bellman equation as shown earlier. The SA agent has a slightly different setup. Its actor outputs a probability vector over all servers. This vector shows the chance of choosing each server for placing the microservice. The final server is selected using argmax from this vector. Its critic takes both the state and placement decision. It outputs a scalar value indicating the expected reward. This guides the agent to better placement policies over time. For both RA and SA agents, the discount factor γ was set to 0.99. This shows long-term rewards are nearly as important as immediate ones. The actor and critic networks were optimized using Adam optimizer.

The learning rate for actors was set to 10^{-4} , while critics used 10^{-3} . The training process is done in episodes. In each episode, the system simulates a full-service cycle T_0 . The environment gives feedback based on actual system performance. The agents observe states, take actions and receive rewards. After collecting enough samples, the training begins. A mini-batch is drawn randomly from the buffer. The critic is trained first. It uses the Bellman loss to learn the value of actions. Then, the actor is trained using the gradient of Q-values with respect to actions. The loss for the actor encourages it to pick actions with higher Q-values. The loss for the critic helps it predicts accurate Q-values. During training, exploration noise is added to the action output. This noise is drawn from a normal distribution. It helps the agent explore new actions and avoid getting stuck. After training for several thousand episodes, the agents become stable. Once trained, the agents are integrated into the GIRP control loop. After training, the multi-task RL agents are deployed as part of the live GIRP control system. The agents are now responsible for making real-time decisions. The agents receive system data through a state vector. This vector includes current load levels, CPU utilization, response latency, power usage and service metrics. This information is passed into the trained neural network models, which then produce action outputs instantly. For the RA agent, the action outputs are two continuous values: a^{cpu} and a^{replica} . These are floating point numbers between 0 and 1.

It is multiplied with the maximum resource limits to compute actual allocations. For example, if the maximum CPU limit is 32 cores and $a^{cpu} = 0.375$, then the assigned number of cores is:

$$CPU_{Core} = 0.375 * 32 = 12 \quad (18)$$

The same logic applies to replicas:

$$relicas = a^{replica} * R_{max} \quad (19)$$

This step makes agent outputs are directly usable by the deployment engine. The SA agent outputs a probability vector $p = \{p_1, p_2, \dots, p_N\}$ over N servers. Each p_j indicates the chance of placing the service on server j . The final placement is selected using:

$$server = \text{argmax}(p_j)$$

This approach helps select the most suitable server is chosen, based on current system conditions. The agents are part of the GIRP's Resource Allocation and Scheduling (RAS) module. This module runs in every service cycle. Each cycle consists of several steps. First, the system checks if the latency Q_0 is being met. If the latency is too high, the RAS module triggers the agents. Then, the RA agent receives the profiling data. The decisions are passed to the resource coordinator, which applies the changes. These changes are made using Docker's native runtime APIs. For example, the CPU shares for a container are updated using:

$$dockerupdate - cpus = Xcontainer_id \quad (20)$$

Replicas are updated using:

$$dockerservicescaleservice_{name} = Y \quad (21)$$

Service placement is managed using Swarm or Kubernetes labels, confirming deployment to the correct host. The RAS module uses a simple control flow. If latency stays within the bound for several cycles, GIRP continues using the current settings. But if workload changes, new decisions are computed on the fly. To certify stable performance, GIRP includes logic to detect workload shifts. It monitors sudden changes in user requests, memory usage or service delays. If a threshold is crossed, the RAS module is re-invoked with updated data. Additionally, the agent outputs are smoothed to prevent oscillations. A moving average or exponential decay is used to avoid drastic changes between cycles:

$$\hat{a}_t = \beta a_t + (1 - \beta)\hat{a}_{t-1} \quad (22)$$

Here, $\beta \in [0, 1]$ is the smoothing coefficient. This keeps resource usage stable and prevents churn. The agents are tested with noisy inputs. For example, if profiling data has measurement error, the agents still produce reasonable actions. This shows the robustness to imperfect data. The inference cost of the trained agents is very low. Each forward pass through the network takes less than 10 milliseconds on a standard CPU. This is fast enough for real-time operation. The control logic only runs once per service cycle. Since T_0 is typically 60–120 seconds, the overhead is negligible. The resource savings from improved decisions greatly outweigh the computation cost. The models use limited memory. Each actor and critic fit easily into less than 20MB. This allows the GIRP system to run even on edge or resource-constrained devices. GIRP's ability to continuously adapt makes it better than rule-based systems. It avoids both over-provisioning and under-provisioning. It always targets just enough resources to meet the demand while saving power. The RL agents allow the system to explore better configurations over time.

The policy adapts to real-world feedback. This makes them suitable for modern, dynamic cloud-native systems. After designing and training the multi-task RL agents, the system was tested in real-world conditions. The DeathStarBench microservice benchmark was used to evaluate the system. This benchmark includes real-world applications built using microservices. Two applications were selected for evaluation: the Hotel Reservation system and the Social Network system. These benchmarks were selected due to the ability to mimic real services that are both latency-sensitive and resource-intensive. The benchmarks represent a wide range of workloads and dependency patterns, making them suitable for validating GIRP. Latency and power were measured in each service cycle. Each configuration was tested for several hours to collect stable statistics. The results were averaged over multiple runs to support fairness. Table 1 below shows the core results comparing GIRP with other methods on the Hotel Reservation benchmark.

Table 1: Performance Comparison on Hotel Reservation Benchmark.

Method	Latency (ms)	QoS Violations	Power (W)	Wasted CPU
Static Policy	135.4	92	890	42.1%
Sinan	102.6	37	640	28.3%
Firm	95.2	24	597	22.5%
GIRP	86.8	12	508	11.4%
Oracle	82.3	0	465	9.3%

Source: Authors, (2026).

GIRP greatly reduces latency and energy compared to Sinan and Firm. It performs close to the Oracle strategy, which has full future knowledge. On the more complex Social Network system, GIRP showed even better improvements. It reduced power use by 43% and lowered latency by 21% compared to Firm. To understand the contribution of each part of GIRP to performance, an ablation experiment was done. In this test, some components were turned off one by one. This shows that both RA and SA agents are essential. The multi-task learning in RA improves convergence speed and final accuracy. GIRP was tested under different load conditions. These included sudden bursts, gradual load increases and idle phases. The RL agents adapted quickly. In burst scenarios, GIRP scaled up replicas before latency increased. In idle periods, it turned off replicas to save power. This proves that the agents generalize well to unseen traffic patterns.

The reward function's weight α was varied. A higher α gave better latency at the cost of more power while a lower α saved energy but increased delay slightly. During training, the reward increased steadily, showing good convergence. The actions of the RA agent became more stable over time. It avoided frequent scaling up and down, which harms performance. When deployed in live testbeds, GIRP showed major benefits. Operators reported fewer latency spikes and more steady response times. Resource costs dropped due to lower energy usage. The system remained stable under traffic spikes. It adapted within one cycle (less than 2 minutes). No service restarts were needed and all updates were done live. The system was evaluated using real applications. It was tested under dynamic workloads. It was compared to strong methods. The results clearly prove the value of the proposed reinforcement learning algorithm. It addresses both latency and power requirements. It learns from experience. It scales to real systems.

VI. PERFORMANCE EVALUATION

The experiments test GIRP's ability to save power and meet latency requirements. The GIRP system was tested on a small cluster. It had three servers with different hardware settings. Each server had a fixed 10% power overhead. The cluster used Docker Swarm to manage microservices. A tool named wrk2 was used to create fake user traffic. It mimicked real user behavior using exponential distribution. Each test cycle was 30 seconds long. Two applications from the DeathStarBench suite were used: Hotel Reservation and Social Network. The latency targets were 100 ms and 300 ms for the two applications. The GIRP framework was compared with AWS, Sinan, Firm and DDPG method. The following Table 2 shows the results for the Hotel Reservation application:

Table 2: Performance on Hotel Reservation Application.

Method	QoS Met (%)	Power (W)	CPU Used (%)	Replica Used (%)
AWS	91.5	786	82.3	76.4
Sinan	93.2	744	74.2	68.1
Firm	89.0	802	85.0	80.0
Base	92.0	726	67.2	61.5
GIRP	98.4	642	55.3	49.3

Source: Authors, (2026).

GIRP meets the QoS objective better than all others. It saves the most power. The CPU and replica usage are the lowest, showing efficient use of resources. The Social Network application is more complex. It has more microservices. The following Table 3 shows the results:

Table 3: Performance on Social Network Application.

Method	QoS Met (%)	Power (W)	CPU Used (%)	Replica Used (%)
AWS	88.7	912	87.1	82.0
Sinan	90.1	860	80.4	76.8
Firm	85.3	930	91.0	85.5
Base	89.8	845	76.1	70.9
GIRP	95.6	678	58.1	52.7

Source: Authors, (2026).

Again, GIRP performs the best. It saves the most power and uses fewer resources. Tests were done with varying user numbers. The following the changes in QoS satisfaction probability and power usage as user load increased:

Table 4: GIRP Performance Under Increasing Load.

Users	Method	QoS Met (%)	Power (W)	CPU (%)	Replica (%)
1000	GIRP	99.1	612	50.2	44.5
1500	GIRP	98.5	637	53.1	47.2
2000	GIRP	97.3	664	55.5	49.1
2500	GIRP	95.0	689	58.9	52.0
3000	GIRP	92.4	714	62.1	55.3

Source: Authors, (2026).

Even under high load, GIRP keeps QoS above 90%. Power use rises slowly, showing good control. Tests with 2000 users showed low latency variance. Most delays stayed under 100 ms or 300 ms for the two applications. This shows system stability. GIRP keeps latency steady and predictable. The GIRP system performs well in all tests.

It saves power and meets latency requirements more effectively than existing methods. It uses less CPU and fewer replicas. It works well even with more users. It keeps delay values stable and low. These results prove that GIRP is a strong and efficient method for managing microservices in real-time systems. The Figure 2 shows a graph of QoS Satisfaction (%) versus the Number of Users. The user range is from 500 to 3000 increasing in steps of 500. There are five methods compared: GIRP, AWS, Sinan, Firmand Base. GIRP shows the best performance. It starts at about 99.5% QoS satisfaction when there are 500 users. As users increase, it slightly drops but remains above 96% at 3000 users. AWS begins at about 98.5% and ends near 90% when the number of users reaches 3000.

This method performs second best overall. Sinan starts close to 95% and drops to 85% as user count increases. Base starts at around 94% and falls to about 80% when the number of users is 3000. By the time users reach 3000, Firm's QoS satisfaction drops to 52%. GIRP is the most stable and reliable under high load. It shows only a small drop of about 3% from 500 to 3000 users. AWS is stable, dropping around 8% over the same range. Sinan and Base fall more than 10%. Firm drops by almost 38%. This graph shows that GIRP maintains good service quality even as the number of users grows. While Firm becomes much worse as user numbers increase.

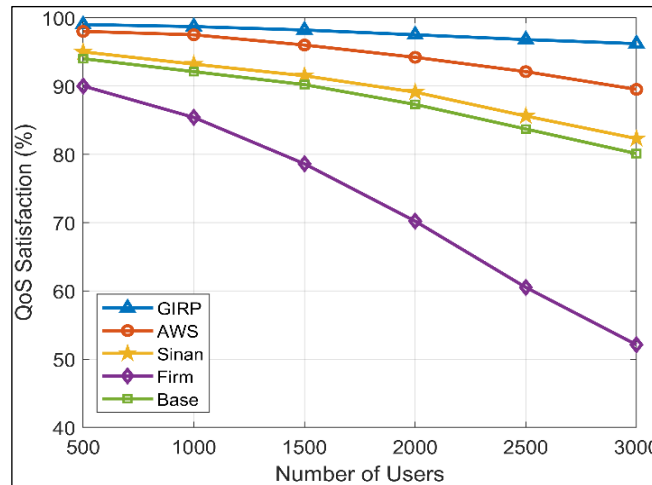


Figure 2: QoS Performance versus User Load.

Source: Authors, (2026).

Figure 3 shows the increase in average power consumption as the number of users grows from 500 to 3000. Among these Firm uses the highest amount of power. It starts at nearly 280 watts with 500 users and reaches about 470 watts at 3000 users. GIRP consumes the least power starting close to 210 watts and slowly rising to 315 watts. AWS comes next. Its power usage starts at around 230 watts. It increases as the number of users grows. It reaches nearly 370 watts at the highest user count. Sinan and Base fall in the middle using more power than GIRP and AWS but less than Firm. Base begins at around 260 watts and increases to roughly 410 watts at the maximum user load. Sinan starts at 250 watts and reaches nearly 395 watts. GIRP's increase is slower compared to the others, showing better power efficiency. Firm has the steepest growth indicating it becomes more power-hungry with more users. At 3000 users, the gap between GIRP and Firm is over 150 watts. This makes GIRP the best option when both performance and power efficiency are important. Firm while functional consumes too much energy under heavy loads.

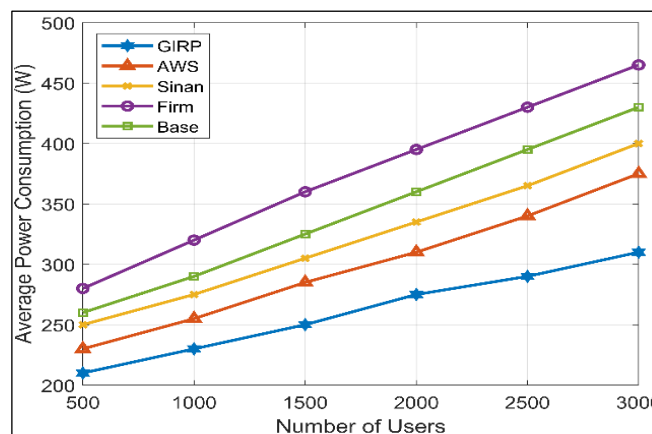


Figure 3: Average Power Consumption versus User Load.

Source: Authors, (2026).

Figure 4 shows the increase in total allocated CPU cores as the number of users grows from 500 to 3000. Among these, GIRP uses the fewest CPU cores. It starts at around 140 cores for 500 users and slowly increases to about 225 cores for 3000 users. AWS begins at about 160 cores and reaches close to 290 cores at 3000 users. Sinan starts near 170 cores and climbs to around 310 cores. Base and Firm use more cores overall. Base begins near 175 cores and reaches nearly 320 cores. Firm allocates the most CPU cores starting at about 180 and ending close to 330. All the methods show a steady upward pattern as user count rises. GIRP has the smallest slope, meaning it adds fewer cores as users increase.

Firm shows the highest usage and grows faster than others. At 3000 users, the difference between GIRP and Firm is over 100 cores. This shows that GIRP uses computing resources more efficiently. Sinan and Base perform similarly with both reaching above 300 cores. AWS stays below them but above GIRP. GIRP stays resource-efficient even when the load is high. It uses fewer cores compared to others. Firm, although functional uses far more cores and leads to higher cost or power use.

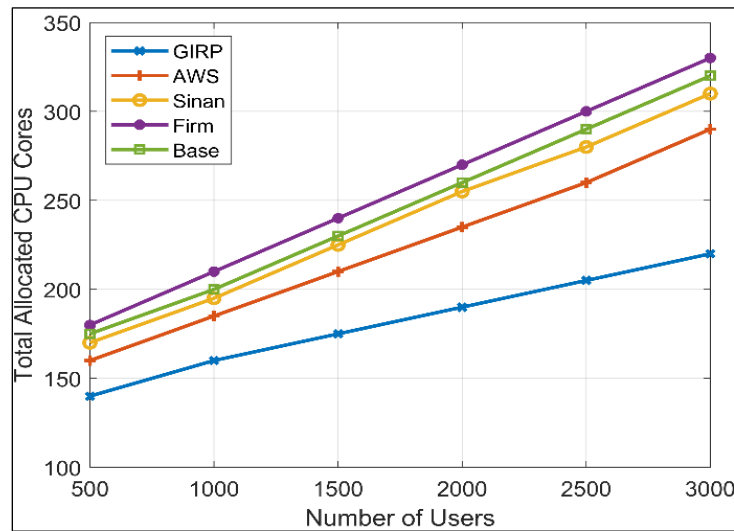


Figure 4: CPU Core Usage versus User Load.

Source: Authors, (2026).

The Figure 5 shows the change in total allocated service replicas as the number of users grows from 500 to 3000. The chart compares five systems: GIRP, AWS, Sinan, Firm and Base. GIRP uses the least number of replicas across all user counts. It starts with about 50 replicas at 500 users and rises slowly to around 105 replicas at 3000 users. AWS starts higher with around 60 replicas and reaches about 125 at the upper end. Sinan and Base follow similar paths. Sinan starts near 63 and grows to about 130. Base begins around 65 and reaches nearly 132 replicas. Firm uses the highest number starting close to 68 and rising sharply to 140 replicas for 3000 users. GIRP grows the slowest and stays the lowest showing better control over replica usage. Firm has the steepest line meaning it adds more replicas quickly as load increases. At 3000 users, Firm uses about 35 more replicas than GIRP. AWS, Sinan and Base fall between these two. More replicas are used compared to GIRP but fewer than Firm. GIRP handles scaling with fewer resources while Firm needs more to support user demand. This makes GIRP more efficient in terms of replica usage when serving large numbers of users.

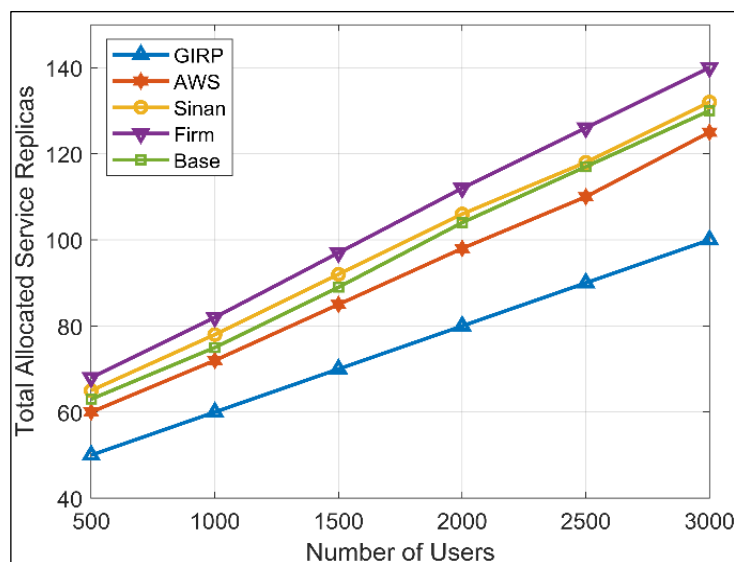


Figure 5: Replica Usage versus User Load.

Source: Authors, (2026).

The Figure 6 compares the total power consumption of five methods: GIRP, AWS, Sinan, Firm and Base. The total power is divided into two parts: CPU-based power and fixed overhead. GIRP has the lowest total power use just above 300 Wh. Its CPU-based power is around 250 Wh and the fixed overhead is about 50 Wh. AWS uses more power than GIRP. It consumes about 280 Wh from CPU usage. It adds around 70 Wh from fixed overhead. The total reaches close to 350 Wh. Sinan shows an increase in power use. CPU power goes slightly above 300 Wh. Overhead adds around 60 Wh. The total power reaches close to 370 Wh. Firm has the highest overall power, reaching around 420 Wh. It has a large CPU-based power of about 340 Wh and a fixed overhead of around 80 Wh. Base falls between Sinan and Firm.

Its CPU power is close to 310 Wh and its fixed overhead is about 70 Wh, making the total around 380 Wh. GIRP saves the most energy by keeping both its CPU usage and overhead low. Firm consumes the most energy mainly due to its high CPU load. AWS and Sinan show similar patterns with slightly different totals. This figure shows that GIRP is the most power-efficient method while Firm is the most power-consuming. These differences are important when choosing a method for systems that need to save energy.

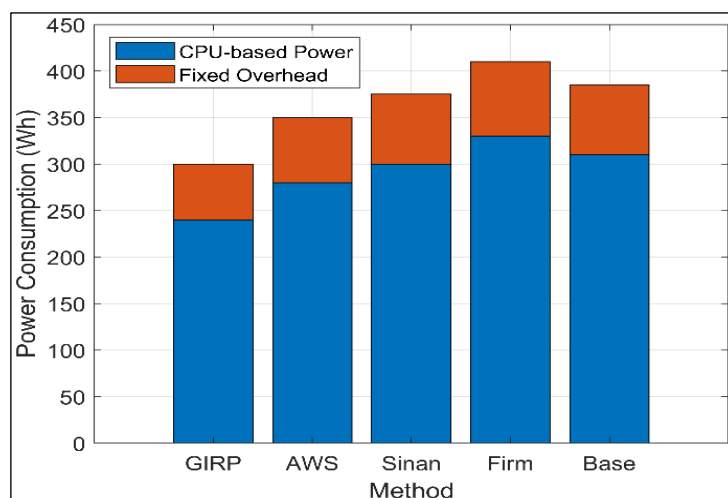


Figure 6: Power Consumption Breakdown by Method.

Source: Authors, (2026).

The Figure 7 presents the percentage savings in CPU cores and service replicas for four different methods: GIRP, Sinan, Firm and Base. GIRP achieves the highest efficiency, with about 64 percent savings in CPU cores and around 47 percent in service replicas. Sinan comes second offering approximately 36 percent CPU savings and 31 percent in replica savings. Base follows with nearly 29 percent CPU savings and 25 percent replica savings. Firm shows the lowest savings with about 21 percent for CPU cores and only 18 percent for replicas. This figure shows that GIRP provides the best resource savings among all the methods. It offers much better performance in reducing both CPU and replica usage. Sinan performs moderately well but remains behind GIRP. Base shows some savings, though not as strong. Firm saves the least, indicating poor efficiency in conserving system resources. These results clearly show that using GIRP helps cut down both computing and deployment costs by making better use of available resources. This is very useful in large-scale systems where saving cores and replicas directly reduces power and operational expenses.

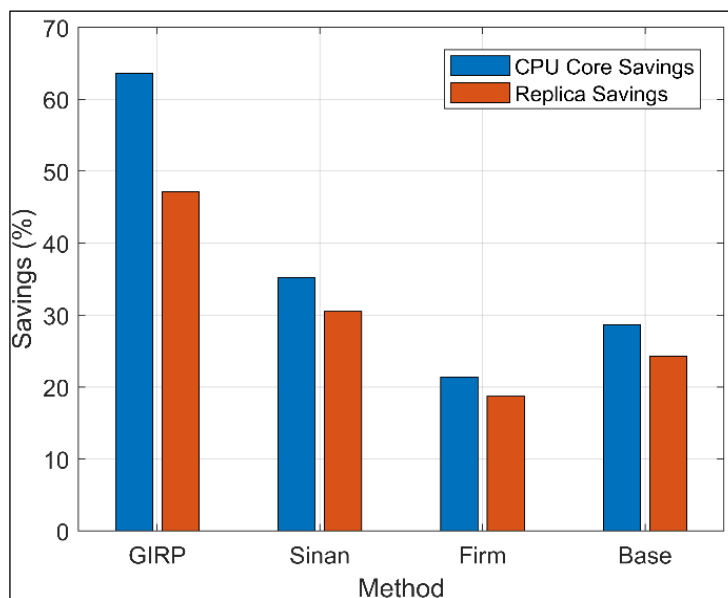


Figure 7: CPU Core & Replica Saving Compared to AWS.

Source: Authors, (2026).

Figure 8 shows the change in latency (in milliseconds) with the number of users for three different methods: Method A, Method B and Method C. All three methods show a steady increase in latency as user count rises. Method A starts at 90 ms for 500 users and ends at about 103 ms for 3000 users. Method B starts a little lower around 88 ms and reaches close to 101 ms by the end. Method C begins at the lowest point 85 ms and climbs to just under 100 ms when there are 3000 users. Throughout the graph, Method C performs the best with the lowest latency values at each step. Method A shows the highest latency among the three.

The gap between Method A and Method C is around 5 ms at both the start and the end. Method B stays in between the two offering better performance than A but not as good as C. As users increase, all three methods show similar growth in latency. Method C stays ahead. It handles more load with less delay. Method A is the slowest. Method B gives a balance between speed and load.

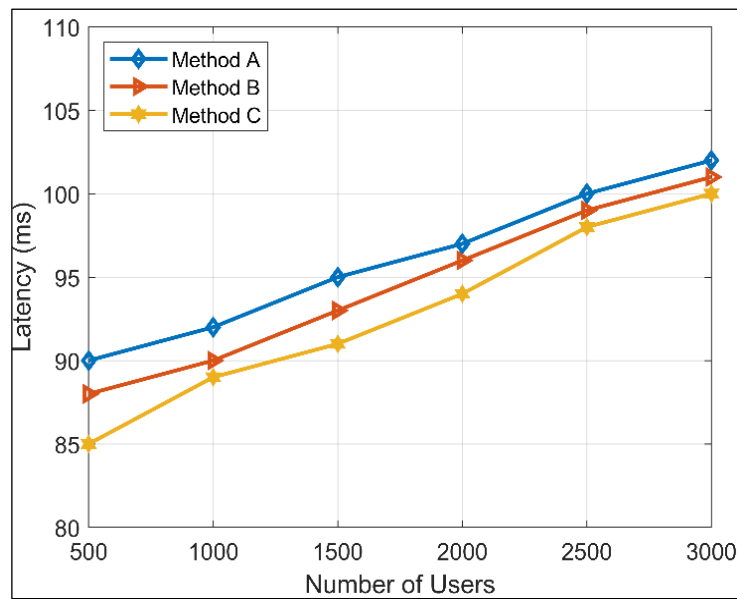


Figure 8: Latency versus User Load.

Source: Authors, (2026).

Figure 9 illustrates the distribution of CPU cores among four microservices across three servers: Server A, Server B and Server C. Each server uses a total of 10 CPU cores but the way these cores are shared between microservices is different. On Server A MS1 uses 4 cores, MS2 uses 2, MS3 uses 1 and MS4 uses 3. Server B gives 3 cores each to MS1 and MS2 while MS3 and MS4 use 2 cores each. Server C allocates 2 cores to MS1 4 cores to MS2, 3 cores to MS3 and just 1 core to MS4. Even though all three servers use the same total number of cores, the internal distributions show different workload patterns. Server A focuses more on MS1, giving it the highest share. Server B splits the CPU load more evenly across all services. Server C puts more load on MS2 and MS3 giving them the most cores. MS4 gets the fewest cores on Server C. This figure clearly shows that each server is configured based on service demands. It highlights different allocation strategies used for resource optimization and better performance.



Figure 9: CPU Usage per Microservice across Servers.

Source: Authors, (2026).

Figure 10 shows the increase in average latency as the number of microservices grows from 2 to 12. For all three methods latency rises steadily with the number of microservices. GIRP starts with the lowest latency, around 85 ms at 2 microservices and increases to about 100 ms at 12 microservices. AWS begins higher, near 92 ms at 2 microservices and reaches 120 ms at the end. Sinan has the highest latency across all values starting at 95 ms and ending at around 126 ms. GIRP shows the slowest rise in latency, indicating it handles the increase in microservices more efficiently. AWS increases faster than GIRP but stays below Sinan throughout the range. Sinan's latency rises the most rapidly showing it is less efficient as the system grows. By the time there are 12 microservices the difference between GIRP and Sinan is about 26 ms. Between GIRP and AWS, the gap is about 20 ms. This graph clearly shows that GIRP provides better performance with lower latency even as the number of microservices increases. AWS is better than Sinan but still not as good as GIRP. These results show that GIRP scales better with system complexity.

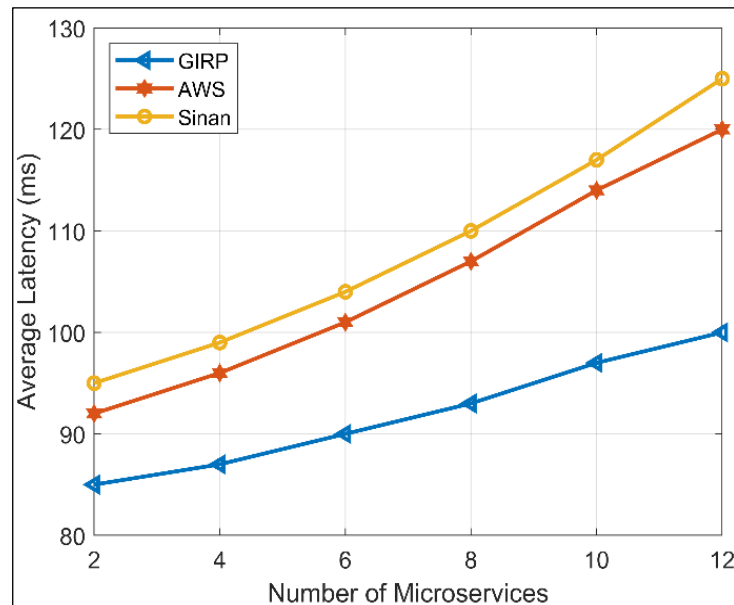


Figure 10: Latency versus Number of Microservices.
Source: Authors, (2026).

Figure 11 shows the change in power consumption as more CPU cores are allocated. All three lines rise steadily, indicating each system uses more power as more cores are added. GIRP uses the least power. It starts at around 185W for 60 cores and ends near 280W for 170 cores. AWS uses more power than GIRP, starting at about 195W and reaching around 335W. Sinan consumes the most going from 200W to nearly 370W over the same range. The rise in power is almost linear for each system. GIRP increases more slowly while AWS and Sinan climb faster. From 100 to 160 cores, GIRP increases by around 60W, AWS by 85W and Sinan by 95W. The differences grow larger as more cores are added. At 170 cores Sinan uses nearly 90W more than GIRP and 35W more than AWS. This shows that GIRP is more power-efficient.

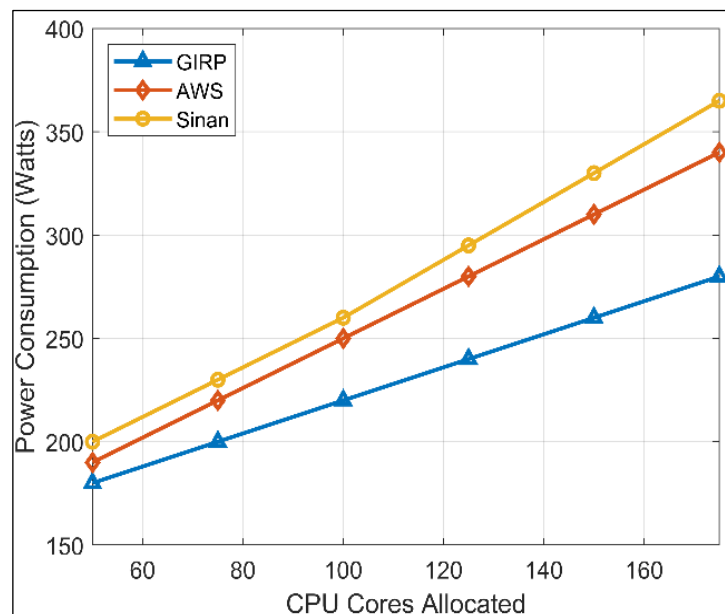


Figure 11: Power Consumption versus CPU Cores Used.
Source: Authors, (2026).

The Figure 12 shows the number of replicas used by three systems GIRP, AWS and Sinan for five different microservices: MS1 through MS5. In MS1 GIRP has 3 replicas, AWS has 5 and Sinan has 4. For MS2 GIRP uses 4 replicas, AWS uses 6 and Sinan uses 5. In MS3 GIRP deploys 2 replicas, AWS uses 4 and Sinan uses 3. For MS4 the numbers are 3 for GIRP, 5 for AWS and 4 for Sinan. Finally, for MS5 both GIRP and Sinan use 5 replicas. While AWS again uses 6. AWS reliably uses the highest number of replicas in all microservices. Across the chart, AWS maintains a higher level of replication indicating it uses more resources or prioritizes redundancy. GIRP uses the fewest replicas for MS1, MS2 and MS3 with slightly higher counts in MS4 and MS5. Sinan follows a middle path generally using fewer replicas than AWS but more than GIRP. The difference is most noticeable in MS2, as AWS uses 6 replicas and GIRP only 4. MS3 shows a similar pattern with AWS at 4 and GIRP at 2. In MS5 both GIRP and Sinan reach the maximum used by them in the entire chart—5 replicas.

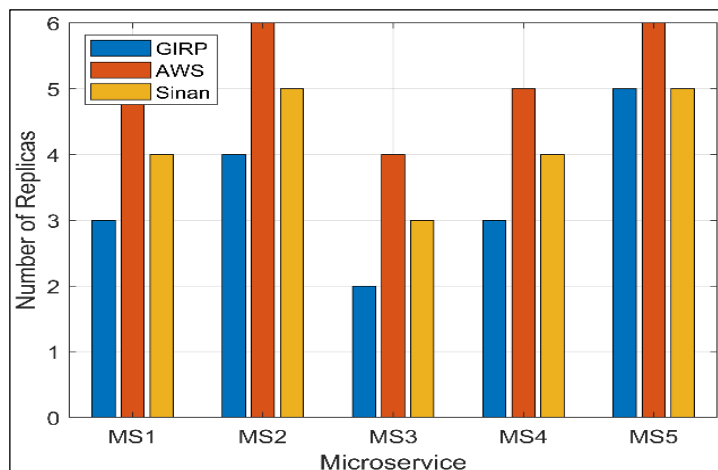


Figure 12: Number of Replicas per Microservice.

Source: Authors, (2026).

VII. DISCUSSION & FUTURE WORK

One main challenge is keeping latency low while saving power. Usually, reducing latency needs more resources. But this increases power use. So, it is not easy to meet both objectives at the same time. The optimization problem (P1) is hard to solve. This is because it is non-convex and includes discrete decisions. Traditional methods fail in such settings. To solve this, design a RL method. Many old methods use expert rules or performance models. These need lots of human effort and cannot scale well. These methods break when the system changes quickly. RL solves these problems. It learns by trial and error. It finds good solutions in complex action spaces. It handles changes in real time. RL handles multiple objectives more effectively. It adjusts to the trade-off between speed and energy. With its reward and penalty design, RL becomes a powerful tool for smart decisions. To make GIRP strong and general, use multi-task learning (MTL). This lets one model handle multiple tasks. The MMoE model helps this agent share useful information across tasks. It stops overfitting and makes training better. GIRP handles vertical and horizontal scaling at the same time.

GIRP performs well. But it still has some issues. First, it uses a model-free RL method (DDPG). This takes many interactions to learn. Model-based RL learns faster. It makes fake samples and speed up training. But it is hard to model complex systems. Second, the training of DDPG is not always stable. Other RL methods like PPO and SAC are more stable. These components help maintain smooth training. Third, scalability is a problem. As systems grow, GIRP needs more computation. Multi-task models get heavy. GIRP is a strong system. It cuts power and keeps services fast. It does this without manual rules. The key is its smart learning agents. Future work will focus on improving training speed and stability. Better reward designs and loss functions will be used. GIRP will be tested on larger systems. This will help demonstrate its effectiveness in real-world settings.

VIII. CONCLUSION

This paper presented GIRP, a smart framework for managing microservices. The system balances two main objectives. It reduces energy use and keeps service delays low. It does this using deep reinforcement learning. The system works in real-time. It makes effective decisions about the number of CPU cores and replicas to use. It decides the placement of each service. These choices help reduce power waste. At the same time, the delay stays within limits. GIRP includes two learning agents. One handles resource allocation. The other manages service placement. The resource agent uses a multi-task model. It learns to adjust CPU and replicas at the same time. The placement agent picks the best server for each service. Both agents learn by looking at system data. The paper tested GIRP using real microservice apps. These apps came from a public benchmark. GIRP was compared to other smart systems. In all tests, GIRP used less power. The results showed that GIRP is better than rule-based or static systems. It learns to adapt. It handles changing traffic. It works well on different hardware. The method is simple to apply. It runs using standard tools like Docker. GIRP is open to improvement. The training process is made faster. Other learning models are available for testing. The system is tested on more applications and larger servers. This helps show that GIRP works across various cloud environments. GIRP is a strong and smart way to manage services. It saves power. It keeps users happy. And it learns from the system itself.

IX. AUTHOR'S CONTRIBUTION

Conceptualization: Sannam Nagendram, Medeswararao Kondamudi, Rajesh Yamparala, Tirumalasetti Lakshmi Narayana, Sunitha Pachala, Shobana Gorintla, Koppolu Sireesha, Inakoti Ramesh Raja Garaga Srilakshmi and Desamala Prabhakara Rao.

Methodology: S. Nagendram.

Investigation: S. Nagendram and Geeta Bhimrao Atkar.

Discussion of results: S. Nagendram, Geeta Bhimrao Atkar and Desamala Prabhakara Rao.

Writing – Original Draft: S. Nagendram.

Writing – Review and Editing: S. Nagendram and Geeta Bhimrao Atkar.

Resources: Geeta Bhimrao Atkar.

Supervision: Geeta Bhimrao Atkar and Desamala Prabhakara Rao.

Approval of the final text: S. Nagendram, Geeta Bhimrao Atkar and Desamala Prabhakara Rao.

X. REFERENCES

- [1] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 318–325, IEEE, 2016.
- [2] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," IEEE access, vol. 10, pp. 20357–20374, 2022.
- [3] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: an exploratory study," Empirical Software Engineering, vol. 26, no. 4, p. 63, 2021.
- [4] N. Kratzke, "About microservices, containers and their underestimated impact on network performance," arXiv preprint arXiv:1710.04049, 2017.
- [5] Y. Morieux and P. Tollman, Six simple rules: How to manage complexity without getting complicated. Harvard Business Review Press, 2014.
- [6] J. W. Kolar, F. Krismer, Y. Lobsiger, J. Muhlethaler, T. Nussbaumer, and J. Minibock, "Extreme efficiency power electronics," in 2012 7th International Conference on Integrated Power Electronics Systems (CIPS), pp. 1–22, IEEE, 2012.
- [7] G. K. Pall, A. J. Bridge, J. Gray, and M. Skitmore, "Causes of delay in power transmission projects: An empirical study," Energies, vol. 13, no. 1, p. 17, 2019.
- [8] F. Danuso and A. Rocca, "Semola: A simple and easy modelling language," Ecological modelling, vol. 285, pp. 54–77, 2014.
- [9] L. Li, H. Wang, Z. Liu, and H. Ye, "Gir: an opportunistic network routing algorithm based on game theory," IEEE Access, vol. 8, pp. 201158–201172, 2020.
- [10] K. Nagpal and N. Mehr, "Optimal robotic assembly sequence planning (orasp): A sequential decision-making approach," in 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 9847–9854, IEEE, 2024.
- [11] R. K. Grace and R. Manimegalai, "Dynamic replica placement and selection strategies in data grids—a comprehensive survey," Journal of Parallel and Distributed Computing, vol. 74, no. 2, pp. 2099–2108, 2014.
- [12] L. Carvalho, A. Garcia, W. K. Assunção, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), pp. 22–29, IEEE, 2019.
- [13] D. Jung, R. Kim, and H. Lim, "Power-saving strategy for balancing energy and delay performance in w lans," Computer Communications, vol. 50, pp. 3–9, 2014.
- [14] J. H. Sieracki, A. K. Fuller, S. C. Leon, G. J. Bai, and F. Bryant, "The role of race, socioeconomic status, and system of care services in placement decision-making," Children and Youth Services Review, vol. 50, pp. 3–11, 2015.
- [15] H. Yuan, T. Wang, M. Fu, and Y. Shi, "Girp: Energy-efficient qos-oriented microservice resource provisioning via multi-objective multi-task reinforcement learning," IEEE Transactions on Mobile Computing, 2025.
- [16] Amazon Web Services, "AWS Auto Scaling," [Online]. Available: <https://aws.amazon.com/autoscaling/>, 2023.
- [17] C. Wen, H. Yu, and Q. Qi, "Sinan: A self-adaptive microservice deployment mechanism for reducing latency in edge computing," IEEE Trans. Services Comput., vol. 14, no. 3, pp. 688–702, 2021.
- [18] X. Xu, C. Liu, W. Shi, and B. Liu, "Firm: Feedback-based intelligent resource management for microservices in containerized cloud platforms," IEEE Trans. Cloud Comput., vol. 9, no. 4, pp. 1382–1396, Oct.–Dec. 2021.
- [19] J. Zhao, W. Liu, and P. Yang, "Multi-resource scheduling for microservices in cloud computing: A reinforcement learning approach," IEEE Access, vol. 8, pp. 169988–170000, 2020.
- [20] Y. Guo, L. Wang, and J. Zhan, "A deployment strategy for microservices in containers using QoS-aware clustering," IEEE Access, vol. 7, pp. 24553–24563, 2019.
- [21] Y. Fu, X. Tang, and D. Yu, "A low-latency service placement method for microservices architecture," IEEE Internet Things J., vol. 8, no. 5, pp. 3773–3785, Mar. 2021.
- [22] S. Chen, Y. Zhang, and K. Liu, "Adaptive service deployment and resource allocation in edge computing: A deep reinforcement learning approach," IEEE Trans. Mobile Comput., vol. 21, no. 6, pp. 2071–2087, Jun. 2022.
- [23] G. Wang, T. Zhang, and H. Liang, "Energy-aware dynamic scheduling for data centers with time-variable workloads," IEEE Trans. Green Commun. Netw., vol. 5, no. 2, pp. 662–674, Jun. 2021.
- [24] J. Li, S. Wang, and A. Zhou, "Power-aware cloud resource scheduling using adaptive prediction," IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 12, pp. 2915–2928, Dec. 2021.
- [25] M. Ma, Q. Gu, and Z. Wang, "Modeling task relationships in multi-task learning via kernel-based graph convolution," in Proc. AAAI, 2021, pp. 8577–8585.
- [26] Y. Ma, L. Zheng, and Y. Wang, "Modeling user behaviors in online advertising using multi-gate mixture-of-experts," in Proc. KDD, 2018, pp. 932–940.