Towards Energy-Efficient Serverless Clouds: A Reinforcement Learning-Based Scheduling Approach

Praewpiraya Wiwatphonthana†, Miika Komu*, Konstantinos Vandikas*,
Lackis Eleftheriadis*, Oleg Gorbatov*, Selome Tesfatsion*, Xuejun Cai* and Roberto Morabito‡
*Ericsson Research, email: firstname.lastname@ericsson.com,
†No affiliation, email: praewpiwwpt@gmail.com, ‡Eurecom, email: roberto.morabito@eurecom.fr

***Abstract*—The serverless computing paradigm has transformed cloud infrastructure by favoring dynamic resource allocation and automatic scaling, reducing the complexity of infrastructure management. However, as workloads with high computing demands and rapid instantiation requirements become more prevalent, energy efficiency remains a critical challenge. Inefficient workload scheduling can lead to suboptimal resource utilization and increased power consumption. In this respect, traditional scheduling policies often struggle to adapt to the dynamic and unpredictable nature of serverless workloads, highlighting the need for more energy-aware scheduling strategies. In this work, we propose a Reinforcement Learning (RL)-assisted scheduling approach to enhance energy efficiency in serverless cloud computing. We develop a Deep Q-Network (DQN)-based scheduler that continuously learns workload placement strategies, minimizing cluster-wide power consumption, and consolidating workload across available Nodes. We implement our approach as a custom scheduling plugin for Kubernetes, ensuring seamless integration with Knative-based serverless workloads. We evaluate our approach using live system telemetry and compare its performance against baseline scheduling techniques. The results show that our RL-based scheduler outperforms the default Kubernetes scheduler by 9.5% in the total cluster CPU consumption, allowing potential energy savings.**

***Index Terms*—Serverless, Scheduler, Energy Awareness, Kubernetes, Knative, Deep Reinforcement Learning**

I. Introduction

Serverless or Function as a Service paradigm provides an abstraction layer to allow executing workloads dynamically without requiring fixed resource allocations [1]–[3] in order to reduce operational costs. However, as cloud workloads grow in complexity, energy efficiency remains a critical challenge, particularly in large-scale cloud data centers, where power consumption is a major concern [4]–[7]. While serverless platforms inherently introduce energy-saving mechanisms, such as suspending idle resources and dynamically scaling functions, workload scheduling remains an open problem for further improving energy efficiency [8]. Inefficient scheduling can lead to suboptimal resource utilization, increased power consumption, and higher operational costs [9].

A key challenge in serverless scheduling is the unpredictability of workloads, which often lack well-defined resource requirements, making workload placement strategies more complex [10]. This unpredictability limits the effectiveness of traditional scheduling strategies, such as Round Robin or Best-fit CPU allocation, which operate with static heuristics that fail to adapt to the dynamic nature of serverless workloads [11]. These scheduling problems are particularly relevant in both cloud and edge computing, where resources may be constrained, requiring more adaptive and energy-aware workload placement strategies.

To address these challenges, machine learning (ML)-based scheduling has emerged as a promising approach [12], [13]. Specifically, reinforcement learning (RL) has been demonstrated potential in dynamically optimizing scheduling policies by continuously learning from system feedback. Hence, we propose a deep Q-Network (DQN)-based scheduling approach that aims to minimize energy consumption while maintaining efficient workload execution. Leveraging RL, our scheduler learns optimal workload placement strategies over time, dynamically adjusting to varying system conditions and resource demands. We evaluate the proposed approach using Knative-based serverless workloads and compare its performance against some baseline scheduling techniques, including Random Scheduling, Round Robin, "Best Fit CPU $< 80\%$", and the Default Kubernetes Scheduler. Our results demonstrate that the RL-based scheduler achieves higher energy efficiency compared to traditional methods by consolidating workloads onto fewer active Nodes while maintaining an acceptable trade-off between performance and energy savings.

Our key contributions are as follows. (1) We propose an RL-based scheduler that optimizes workload placement for energy efficiency in serverless cloud environments. (2) We implemented our solution as a custom Kubernetes scheduling plugin, enabling seamless integration with Knative-based serverless workloads. To the best of our knowledge, this is the first work to address energy-aware scheduling for Knative. (3) We benchmark our approach against multiple scheduling baselines using live system telemetry, analyzing its impact on energy consumption, resource utilization, and response time.

The paper is organized as follows. Section II provides a background on serverless computing, cloud orchestration, and scheduling approaches. Section III describes the system design and architectural components of the proposed scheduling framework. Section IV details the DRL agent and the decision-making mechanism. Section V presents the experimental setup, and Section VI discusses our results and findings with respect to the baseline comparison. Section VII highlights related work, positioning our contributions within

the broader research landscape. Finally, we conclude the paper with a summary of findings and directions for future research.

II. Background

Serverless Computing (SC) is a cloud service model that allows one to build and run applications without managing underlying servers or infrastructure. In this model, developers can focus on the code logic, while all the responsibilities of provisioning, scaling, and maintaining these servers are handled by the cloud service provider (CSP) [14]. A key feature of SC services is the use of serverless functions, where each function only runs for the time necessary to complete the task; once execution is finished, resources are automatically released. To support serverless computing, serverless platforms are specifically designed to manage the execution of serverless functions [10], [15], [16]. These platforms are widely adopted to simplify application development and their cost-effective pay-as-you-use billing model. Serverless platforms [17] from leading CSPs and open-source platforms [18] are available to offer flexibility and control in container-based environments [19].

One of the key concepts of SC is Cloud Orchestration that allows automating the deployment, management, and scaling of containerized applications in distributed environments. One of the most widely adopted platforms is Kubernetes. Kubernetes supports the *declarative configuration* that allows developers to define the deployment based on the controller design pattern, while the platform automatically maintains it. Kubernetes consists of several core components that are divided between the *Control Plane* and *Worker Nodes*. The *Control Plane* is responsible for managing both the *Worker Nodes* and the *Pods* across the cluster. The *Control Plane* manages the overall state of the Kubernetes cluster and coordinates the scheduling and execution of containers.

Knative is one of the open-source alternatives to support serverless operations in Kubernetes maintained by CNCF. Knative simplifies building, deploying, scaling, upgrading, and managing serverless applications and event-driven functions. Although Knative has three key components, which are Function, Serving, and Eventing, Knative's components are independent by design [20]. *Knative Function* is a programming model for developing and deploying serverless functions. *Knative Serving* manages and controls the deployment, upgrading, versioning, routing, and scaling of serverless applications. *Knative Eventing* is a framework for building event-driven architectures, allowing services to communicate asynchronously and respond to specific triggers. Monitoring tools such as Prometheus and Kubernetes Efficient Power Level Exporter (*Kepler*) [21], which is a *Prometheus* exporter have been developed to improve observability in containerized environments. Prometheus tracks resource usage and performance. Kepler focuses on estimating power consumption using machine learning models based on CPU performance and kernel traces.

Kubernetes Scheduler is responsible for distributing workloads across Nodes in a cluster and allows one to identify a *feasible Node*, which satisfies the specific requirements to run a newly created or unscheduled Pod. Kubernetes has *kube-scheduler* as a default scheduler that runs as part of the Control Plane. To manage Pod scheduling, the scheduler works alongside the Kubernetes API Server that acts as a central communication hub for all Kubernetes components. When a new Pod is created, the API Server adds it to the queue [22] and the scheduler then performs *filtering* and *scoring*: In *Filtering*, Nodes that do not meet the Pod's requirements are filtered out and it results in a list of *feasible Nodes*. In *Scoring*, the *feasible Nodes* are scored based on criteria such as resource efficiency and affinity rules. The Node with the highest score is selected for the placement of the Pod and, finally, the scheduler binds the Pod to the selected Node and updates the Pod state to indicate on which Node it will run on [23], [24].

The Scheduling Framework (SF) represents a modular architecture integrated within the Kubernetes scheduler, improving its adaptability and extensibility [25]. SF provides a set of *plugin APIs* that allow most scheduling functionalities to be implemented as plugins for the core scheduler. The scheduling can be customized by developing a standalone scheduler or extending the default *kube-scheduler* using Scheduler Extender (SE) [26], [27]. Although SE enables rapid extensions with minimal setup, it is limited to specific extension points and relies on HTTP communication, which introduces additional overhead. To address these limitations, SF provides plugin-based extensions [28]. These are compiled directly into the scheduler binary, eliminating inter-process communication overhead and allowing finer control over scheduling phases.

III. System Design

Fig. 1 provides an overview of the implementation architecture and its components. **Kubernetes Control Plane** is a core infrastructure responsible for managing all major deployed components, including the Serverless Platform, Scheduler, Scheduler Controller, and Metrics Collector. We employ **Knative** Functions in our system. The API requests are first queued through the Event Queue and then forwarded to the **Kubernetes API Server** by the Dispatcher. **Scheduler Controller**, developed here, employs various algorithms to optimize workload placement decisions. It is capable of functioning as an RL agent or as a simpler baseline algorithm (Random, Round Robin, and Best Fit). The controller learns from historical data to improve placement strategies.

The **Data Manager** is developed by us; it records the state of the cluster in the Redis database, including metrics on resource usage and availability, to support the Scheduler Controller. The Data Manager scrapes resource usage and Pod CPU consumption metrics from cAdvisor in 15 second intervals. **Redis** is an in-memory data store, and we utilize it for data sharing between the Scheduler Controller and Data Manager based on its publish-subscribe mechanism. **Metrics Collector** gathers real-time metrics using Prometheus, Kepler, and cAdvisor. **Prometheus** aggregates resource usage data

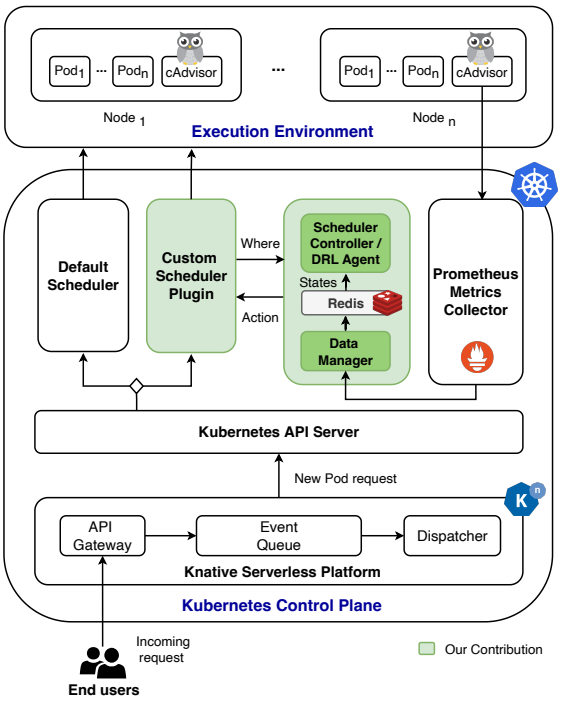

Fig. 1. System Architecture Implementation Overview

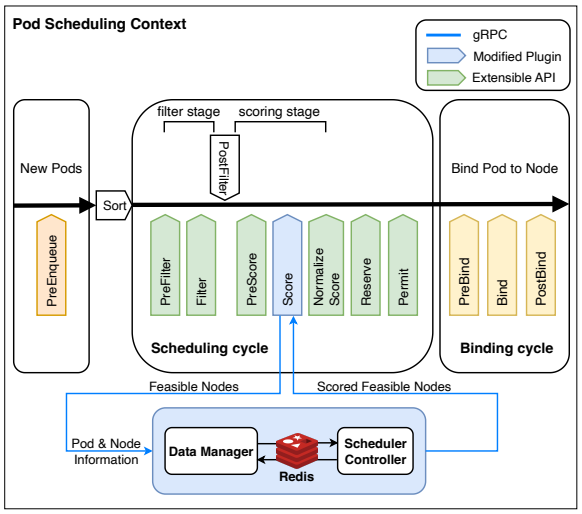

Fig. 2. Kubernetes Scheduling Framework Implementation Overview

across the cluster, **cAdvisor** monitors container-level statistics such as CPU and memory usage, and Kepler provides energy consumption metrics at the Node level.

In our implementation, we deploy a custom scheduling plugin as an additional second scheduler alongside the default Kubernetes scheduler. The **default scheduler** continues to manage scheduling of non-serverless Pods. The **custom scheduling plugin** is designed specifically to handle serverless workloads. As shown in Fig. 2, we have modified *score* processing stage of the scheduling plugin (highlighted in blue in the figure) to introduce custom logic for serverless workload placement. We have configured the custom scheduler to be triggered only in the case of Knative workloads, but other workloads are routed to the default scheduler.

When a new API call to serverless function is invoked, the Knative system first checks if a serving Pod is already running. If such a Pod exists, then the call is routed to the corresponding Pod. Otherwise, Knative requests Kubernetes to start a new Pod. This, in turn, invokes the custom scheduler to determine a Node where to start the workloads as shown in Fig. 2. Our custom plugin receives the invocation and delegates the Node placement decision to the Data Manager by sending information about Pods and available Nodes over gRPC. Next, the Data Manager further requests the Scheduler Controller (by publishing via a Redis database) to compute optimal placement for the new Knative serving Pod based on RL or some other more rudimentary logic. Then, the Scheduler Controller calculates the placement scores based on cluster state metrics for all available Nodes in Kubernetes and communicates this information back to the Data Manager that further delivers them to the custom scheduling plugin. Finally, the custom scheduler completes the scoring stage, assigning the serving Pod to the Node with the highest score.

IV. Deep Reinforcement Learning Agent

RL [29] is a ML approach, where an agent learns to maximize cumulative rewards through trial and error, guided by environmental feedback.

Deep Q-Learning (DQN) [30] is a popular algorithm that overcomes Q-learning's [31] limitations by using a deep neural network to estimate Q-values instead of tables, enabling scalability in large state spaces. DQN is used by the Scheduler Controller.

DQN is characterized by three key components: State Space (S_t), Action Space (A_t), and Reward (R_t). In the following, we describe these in detail:

- **State Space**: contains 7 state features for each node, thus resulting in 28 features (4 nodes). These are Node-Level metrics and Cluster-Level Metrics. Node-Level metrics are CPU and Memory Usage (%), CPU Availability (%), Memory Availability (%) and Number of Pods running on Node. Cluster-level metrics are Total Cluster CPU and Memory Usage (%). Metrics measured as a percentage (i.e. CPU/memory usage) are binned into ranges of 0..9. Then these and all other metrics are normalized between 0..1 via MinMax normalization.
- **Action Space**: Since we consider 4 Nodes, 4 actions are possible, indexed as [0, 1, 2, 3] representing the index of the Node where the serverless workload will be placed.
- **Reward Function**: There are two main design goals behind this function: 1) reduce the overall cluster CPU usage and 2) maximize Node usage. Reward is defined in 1:

$$Reward = \frac{\text{Node-specific CPU Usage Rate \%}}{\text{Cluster CPU Usage Rate \%}} - p \quad (1)$$

Where:

- **Node-specific CPU Usage Rate (%)**: refers to the average utilization rate of the Node chosen by the agent's action during the episode.
- **Cluster CPU Usage Rate (%)**: represents the average utilization rate of all Nodes in the cluster.
- **p**: Penalty is subtracted to account for any undesirable conditions during the episode.

Penalty p discourages under-utilization and prevents over-saturation of Worker Nodes with serverless workloads. Thresholds such as 70% and 80% CPU utilization are considered based on [32] and [33]. When these thresholds are crossed, bottlenecks may occur which can degrade performance. 80% was chosen to maximize resource allocation 2.

$$p = \begin{cases} -\alpha \cdot \ln(1 + \frac{\text{CPU Usage}}{k}) * s & \text{CPU Usage} \leq 80\% \\ \alpha \cdot e^{\beta \cdot (\text{CPU Usage} - 80)} & \text{CPU Usage} > 80\% \end{cases} \quad (2)$$

Where:

- α **and s**: These are scaling factors that control the severity of the penalty.
- **k**: A constant used to adjust the penalties growth.
- β: A constant that controls the growth rate of the exponential value.

The training process consists of 350 episodes, each with 14 steps, and 40 substeps per step. Episode structure is illustrated in Fig. 3. In each step a Pod is created for a serverless function (i.e. fib, Fibonacci calculation). The sequence of functions is shuffled to ensure diverse task placement. When a step begins, a request is made to create a Pod and invoke the function. The Knative platform handles this either by requesting a new Pod or by routing the request to a Pod that is already running. Each step is followed by a 15-second delay to ensure that all changes in the cluster are reflected.

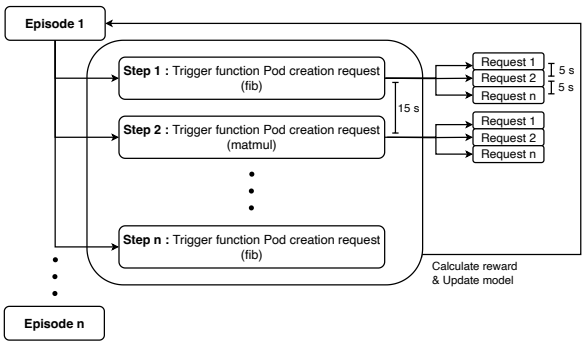

Fig. 3. Episode Structure

V. Experimental Setup

This section details the experimentation setup, including the cluster configuration and serverless workloads we used.

A. Cluster Setup

We deployed a Kubernetes cluster (v1.29.10) on five virtual machines (VMs) running in an OpenStack based cloud. Each VM runs Ubuntu 22.04 LTS (Jammy Jellyfish) as its operating system. The cluster consisted of one Control Plane Node and four Worker Nodes, with detailed specifications provided in Table I. The Knative serverless platform (v1.16.0) was deployed on the Control Plane Node to manage serverless workloads. The Control Plane Node orchestrates cluster operations and scheduling, while the Worker Nodes execute the Knative workloads.

TABLE I
Cluster Node Specifications

Node Type	CPU	RAM	Storage
Control Plane	30 vCPUs	214.8 GB	1000 GB
Worker (×4)	4 vCPUs	16 GB	40 GB

The prototype environment is built using Go for developing a custom Kubernetes scheduler plugin and Python for implementing the Scheduler Controller and Data Manager modules. PyTorch is used for RL model training, while gRPC and Protocol Buffers facilitate low-latency communication. Redis provides in-memory caching and real-time pub/sub messaging, and Asyncio supports asynchronous operations in the Scheduler Controller and Data Manager modules.

B. Serverless Workloads

The workloads used in this study consist of multiple CPU- and memory-intensive functions. These workloads are implemented using Knative Functions [34], which provide templating for serverless function creation. These templates allow users to specify the programming language and invocation format. In this work, Python is chosen as the implementation language, while HTTP is used as the invocation format.

Our study considers four serverless functions: `fib` (Fibonacci) recursively computes Fibonacci numbers, `matmul` (Matrix Multiplication) generates two matrices and performs matrix multiplication, `mattran` (Matrix Transpose) generates a matrix and transposes it, and `float` executes a set of floating-point operations (sin, cos, sqrt).

The resource requirements of each function varies, categorized as low ([+]), medium ([++]), high ([+++]), or negligible ([-]) based on observed resource consumption in our experimental environment. Each function is duplicated multiple times per episode to simulate serverless execution patterns. These functions are shuffled in each episode to create diverse workloads for the scheduler. Table II summarizes the details of workloads, including data range and increment size. Input data, provided in JSON format, are shuffled per episode to introduce variation. Input data are provided in JSON format, with predefined data ranges shuffled per episode to introduce variation. To ensure reproducibility and maintain distinct workload distributions across phases, different random seeds are used for training and evaluation episodes.

TABLE II
SERVERLESS WORKLOAD DETAILS

Function	Res. Req		Dups	Data Range	Incr.
	CPU	Mem			
fib	[+++]	[-]	6	28 to 34	1
matmul	[++]	[+]	3	1000 to 3000	500
float	[+]	[++]	3	500000 to 3000000	500000
mattrans	[++]	[+++]	2	6000 to 18000	3000

VI. EVALUATION AND DISCUSSION

In this section, we present an evaluation of our approach and discuss the results. We first introduce the evaluation cases and the associated metrics. Then, we demonstrate the experimental results, including workload placement and system utilization, energy consumption, and application performance. Throughout this section, the term *evaluation period* refers to the timeframe from the start of the evaluation run to the completion of all workloads in the final episode.

A. Baselines Comparison and Evaluation Metrics

For each scheduler, the evaluation performed on our testbed consisted of 20 episodes, each comprising 14 steps, each step having 40 substeps (number of requests). We compare the performance of the DQN-based scheduler against four baseline scheduling strategies commonly used in serverless and cloud computing environments:

- **Random** serves as a naive baseline, where workloads are assigned to a randomly selected Node without considering system conditions or resource utilization.
- **Round Robin** follows a cyclic selection process, distributing workloads sequentially across available Nodes.
- **Best Fit CPU $<$ 80%** restricts workload placement to Nodes with CPU utilization below 80%. Among eligible Nodes, it prioritizes the one with the highest CPU utilization, with the aim of maximizing resource utilization while preventing Node overload.
- **Kubernetes Default Scheduler** follows a two-phase scheduling process. In the *filtering phase*, the scheduler selects Nodes that meet the workload's resource requirements. Then, the *scoring phase* ranks the feasible Nodes based on predefined policies, ultimately selecting the highest scoring Node for workload placement. This scheduler policy is widely used in Kubernetes environments but lacks built-in mechanisms for energy-aware scheduling, making it an important baseline for evaluating our RL-based approach.

Using a comparison with these baselines, our aim is to demonstrate the suitability of our approach in optimizing energy efficiency, resource utilization, and workload placement.

To assess the performance of the proposed scheduling strategies, we consider multiple evaluation metrics. These factors capture different aspects of system performance, including workload placement, energy efficiency, and application responsiveness.

Workload Placement and System Utilization. We track the *placement decision count*, which provides insights on the frequency of workload placement patterns to analyze how workloads are allocated across cluster Nodes. This metric helps in understanding how the scheduler distributes workloads and utilizes resources across available Nodes.

Energy Consumption Metrics. We monitor *cluster CPU usage rate*, which represents the average CPU utilization across all cluster Nodes on a per-second basis. Based on our design, this metric is used as a primary indicator of energy consumption in our system. *Kepler cluster power consumption* estimates the total power consumption of the cluster in watts. Furthermore, we assess *Node CPU usage rate*, which measures the per-second average CPU utilization at the individual Node level over a specified time range.

Application Performance Metrics. We evaluate the impact of scheduling decisions on application performance with *response time*, which is defined as the average time required to complete a serverless task.

B. Empirical Results

Placement Decision Count. We evaluated each scheduler over 20 episodes, during which Kubernetes made a total of 280 scheduling decisions per case. The Node placement decisions for each scheduling strategy is illustrated in Fig. 4.

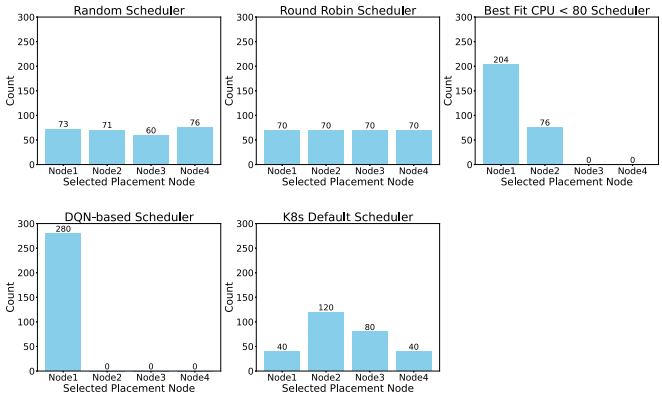

Fig. 4. Workload Placement Decision across Episodes

The Random Scheduler placements are relatively uniformly distributed across the Nodes with minor variations reflecting the inherent randomness. The Round Robin Scheduler follows a uniform distribution as the algorithm cycles sequentially through the Nodes. In contrast, the Best Fit CPU $<$ 80% Scheduler utilized only two Nodes, as it prioritized placing workloads on the most utilized Node while maintaining CPU usage below 80%. The DQN-based Scheduler concentrated all placement decisions on a single node. This indicates that the trained model optimized for specific metrics, such as overall cluster CPU utilization, by consolidating workloads onto fewer Nodes. Meanwhile, the Kubernetes Default Scheduler showed some placement variability, with a bias toward certain

Nodes influenced by resource availability and scheduling policies.

Cluster CPU Usage Rate. Cluster energy efficiency is analyzed through CPU utilization in different scheduling strategies, sampled at 30-second intervals from Prometheus. The Random Scheduler demonstrated the highest cumulative CPU usage, possibly because it does not account for any resource availability or workload conditions. Similarly, the Round Robin Scheduler, which distributes workloads sequentially without considering any resource constraints, also exhibited high Cluster CPU usage. The Default Kubernetes Scheduler followed a similar pattern, with identical CPU usage to the Round Robin Scheduler, indicating that its placement decisions did not significantly improve CPU efficiency.

In contrast, the Best Fit CPU $<$ 80% Scheduler reduced CPU usage by 4.41% compared to the Random Scheduler by consolidating workloads while maintaining the threshold. The DQN-based Scheduler achieved the lowest CPU usage, reducing utilization by 9.71% compared to the Random Scheduler and 9.42% compared to the Default Kubernetes Schedule, showcasing its capability to optimize workload placement and reduce overall cluster CPU utilization. We also compared CPU utilization with power consumption estimations from Kepler and both showed similar trends and aligned well with each other. However, Kepler does not yet measure CPU frequency scaling well [35], so we did not base our measurements on it.

Node CPU Usage Rate. We also analyzed Node-specific CPU usage patterns. The Random Scheduler and Round Robin Scheduler distributed CPU usage relatively evenly across all Nodes. In contrast, the Best Fit CPU $<$ 80% Scheduler consolidated workloads, leading to the highest CPU utilization on Node 1, moderate activity on Node 2, and minimal usage on other Nodes due to the policy of the best-fit scheduler. The DQN-based Scheduler consolidates the workload even further, with Node 1 dominating CPU usage while other Nodes remain nearly idle. Meanwhile, the Kubernetes Default Scheduler displayed varied CPU usage patterns, with specific Nodes (e.g., Node 2) being preferred over others.

Response Time. Fig. 5 illustrates the average response time of serverless functions across various scheduling strategies. The fluctuation in response time observed across different tasks reflects the varying computational intensity of serverless functions, where more resource-intensive tasks naturally result in higher average response times. Most strategies achieve comparable response times, except the DQN-based scheduler, which has slightly higher response times. This increase could be attributed to the additional time required for decision computation using the DQN model or CPU saturation caused by consolidating workloads onto a single Node, which increased the likelihood of resource contention.

C. Discussion

The empirical results highlight the advantages of RL in optimizing workload placement for energy efficiency. The

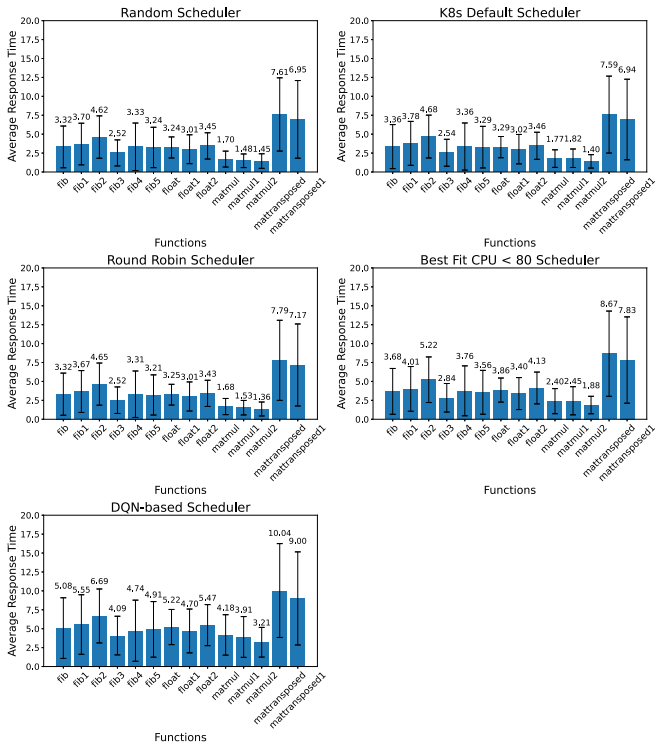

Fig. 5. Serverless Workload Function Response Time

DQN-based Scheduler demonstrates a clear preference for workload consolidation, efficiently allocating tasks to a single Node. This strategy minimizes overall cluster CPU usage and aligns with Kepler's power consumption estimates, suggesting significant energy savings.

Compared to other scheduling strategies, Best Fit CPU $<$ 80% also consolidates workloads but follows a predefined CPU threshold, leading to a more efficient Node utilization. The Random and Round Robin Schedulers, on the other hand, distribute workloads evenly across Nodes, which results in higher CPU usage but prevents localized resource contention. The Kubernetes Default Scheduler exhibits variability in Node selection, reflecting its two-phase scheduling policy that factors in resource availability and predefined scoring criteria.

One trade-off observed with the DQN-based approach is the higher response time variability, particularly for computationally intensive functions (Fig. 5). This can be attributed to CPU saturation on the selected Node, which may introduce resource contention under high workload intensity. In this context, performance stability refers to maintaining consistent response times across varying workloads without excessive fluctuation. Although this impact is moderate, it highlights the need for adaptive mechanisms that balance workload consolidation with stability. Future work could explore hybrid RL-based strategies that dynamically adjust scheduling decisions based on workload conditions. While not empirically evaluated in this study, such strategies could mitigate potential latency increases by introducing dynamic CPU thresholds

or adaptive load redistribution, preserving energy efficiency while improving response time consistency.

Overall, this empirical analysis strengthens the potential of RL-based scheduling in serverless cloud environments, particularly for applications prioritizing energy savings. Additionally, through the integration of dynamic adaptation mechanisms, future RL-based approaches can further enhance scheduling efficiency while minimizing trade-offs.

Other types of RL algorithms can be considered in the scope of this work such as Policy Gradient Methods (PGM) [36] and Actor-Critic [37]. They optimize directly the policy by mapping states to actions as opposed to DQN which estimated the expected reward using the Q-Function. As such, PGM can be a better fit for continuous action spaces. In the scope of this work, the action space has been discretized in the range of [0..4] for simplicity. However, if we consider a much large set of possible Nodes to deploy the serverless workload, then a continuous action space might be more preferable. The actor critic is an alternative approach that makes use of an actor or a module that makes a decision based on a policy and the critic which evaluates that decision. From this point, actor-critic can be seen as an improvement to Policy gradient and as such it is also suitable for continuous action spaces.

VII. Related work

Scheduling plays a pivotal role in serverless resource management, determining where and how functions are placed within the cloud infrastructure to meet criteria such as Quality of Service (QoS), resource utilization, and energy efficiency [38].

Adeppady *et al.* [39] proposed a scheduling framework which utilizes a threshold-based queueing mechanism to manage the cold, warm, and running states of containers, aiming to reduce server energy consumption while maintaining QoS targets. Additionally, Aslanpour *et al*. [40] developed Faashouse, an energy-aware resource scheduling algorithm that minimizes consumption by using computation offloading to manage energy imbalances across nodes. Arys and Carlier *et al*. [41] developed an energy-aware scheduling method using affinity models based on offline profiling of serverless functions. Rastegar *et al*. [42] created an energy-aware scheduler employing linear programming to minimize energy consumption while meeting deadlines.

Multiple literature surveys have explored the application of ML for resource scheduling in cloud computing environments. These works highlight how ML techniques can dynamically adapt to workload fluctuations, optimize resource utilization, and enhance energy efficiency [12], [13], [43], [44].

Particularly, some ML-based scheduling works for improving energy saving in Kubernetes are also proposed. Rothman and Chamanara [45] developed RLKube, a Kubernetes scheduling plugin leveraging Double DQN (DDQN) with Prioritized Experience Replay (PER) to optimize resource utilization, Pod throughput, and energy efficiency. Ryuki Douhara *et al*. [46] developed the Workload Allocation Optimizer (WAO), integrating a scheduler and load balancer with neural network models to optimize power consumption and response time, achieving an 8% reduction in power consumption compared to the default Kubernetes scheduler. Chiorescu *et al*. [47] have integrated a random forest classifier-based model into the Kubernetes scheduler, which is evaluated in an OpenFaaS-based environment.

However, integrating RL for energy-aware scheduling in serverless environments remains relatively unexplored. Our proposed approach addresses this gap by leveraging RL-based methods to optimize energy efficiency while maintaining workload performance in serverless systems.

VIII. Conclusions

In this paper, we proposed an RL-assisted scheduling approach to enhance energy efficiency in serverless cloud environments. Addressing the limitations of traditional scheduling strategies, our approach leverages DQN-based learning to dynamically optimize workload placement. Unlike heuristic-based methods that rely on predefined rules, our scheduler adapts to real system conditions, improving resource utilization while minimizing overall cluster power consumption.

To validate our approach, we implemented it as a custom Kubernetes scheduling plugin, ensuring seamless integration with Knative-based serverless workloads. In our extensive empirical evaluation in a Kubernetes cluster, the DQN-based scheduler is compared against four baseline scheduling strategies. Our results demonstrate that the DQN-based scheduler achieves the lowest cluster-level CPU usage. Our scheduler uses 9.5% less CPU than the Default Kubernetes Scheduler when the total CPU consumption of the cluster was measured. It highlights its potential to improve energy efficiency. The findings also reveal a trade-off in response time, where workload consolidation on a single Node may lead to resource contention, underscoring the need for adaptive mechanisms that balance energy savings with performance stability.

We believe that our work contributes to the growing area of intelligent scheduling for serverless and edge computing environments. Integrating RL into workload orchestration represents a step toward more autonomous and resource-efficient cloud infrastructures.

In future work, we aim to measure power draw at the hardware level to obtain more accurate insights into energy consumption. However, a lower power draw does not always imply lower total energy use, since reduced power may prolong task execution. We will also investigate whether the higher response time observed with the DQN scheduler arises from this effect. As DQN is relatively heavy-weight for small clusters, we should examine its scalability on larger clusters and more complex scenarios with mixed workloads (serverless and non-serverless) possibly including I/O bound workloads. It would be useful to evaluate alternative RL algorithms with additional metrics and benchmark their energy consumption.

REFERENCES

[1] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless Computing: State-of-the-Art, Challenges and Opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[2] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on Serverless Computing," *Journal of Cloud Computing*, vol. 10, pp. 1–29, 2021.

[3] G. Adzic and R. Chatley, "Serverless Computing: Economic and Architectural Impact," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 884–889.

[4] IEA, "Executive Summary–Electricity2024 Analysis," https://www.iea.org/reports/electricity-2024/executive-summary/, accessed: 2024-19-11.

[5] ——, "Digitalization and Energy – Analysis." https://www.iea.org/reports/digitalisation-and-energy/, accessed: 2024-19-11.

[6] A. Alhindi, K. Djemame, and F. B. Heravan, "On the Power Consumption of Serverless Functions: An Evaluation of OpenFaaS," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2022, pp. 366–371.

[7] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward Sustainable Serverless Computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[8] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE international conference on cloud engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[9] M. Kumar, S. C. Sharma, A. Goel, and S. P. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *Journal of Network and Computer Applications*, vol. 143, pp. 1–33, 2019.

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

[11] M. R. Garey and D. S. Johnson, "" Strong " NP-Completeness Results: Motivation, Examples, and Implications," *Journal of the ACM (JACM)*, vol. 25, no. 3, pp. 499–508, 1978.

[12] G. U. Srikanth and R. Geetha, "Effectiveness review of the machine learning algorithms for scheduling in cloud environment," *Archives of Computational Methods in Engineering*, vol. 30, no. 6, pp. 3769–3789, 2023.

[13] W. Khallouli and J. Huang, "Cluster resource scheduling in cloud computing: literature review and research challenges," *The Journal of supercomputing*, vol. 78, no. 5, pp. 6898–6943, 2022.

[14] G. Cloud, "What is serverless computing," https://cloud.google.com/discover/what-is-serverless-computing?hl=en, accessed: 2024-08-10.

[15] M. Bensalem, F. Carpio, and A. Jukan, "Towards Optimal Serverless Function Scaling in Edge Computing Network," in *ICC 2023-IEEE International Conference on Communications*, 2023, pp. 828–833.

[16] A. Mampage, S. Karunasekera, and R. Buyya, "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[17] AWS Lambda: https://aws.amazon.com/lambda/, Google Cloud Functions: https://cloud.google.com/functions/, Azure Functions: https://azure.microsoft.com/en-gb/products/functions/, IBM Cloud Code Engine: https://www.ibm.com/products/code-engine.

[18] OpenFaaS: https://www.openfaas.com/, OpenWhisk: https://openwhisk.apache.org/, Fission: https://fission.io/, Kubeless: https://github.com/vmware-archive/kubeless, Knative: https://knative.dev/docs/.

[19] A. Palade, A. Kazmi, and S. Clarke, "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642. IEEE, 2019, pp. 206–211.

[20] J. Chester, *Knative in Action*. Simon and Schuster, 2021.

[21] Kubernetes efficient power level exporter (kepler). Accessed: 2024-15-09. [Online]. Available: https://sustainable-computing.io/

[22] Kubernetes Community, "Scheduler queues," https://github.com/kubernetes/community/blob/f03b6d5692bd979f07dd472e7b6836b2dad0fd9b/contributors/devel/sig-scheduling/scheduler_queues.md, accessed: 2024-11-08.

[23] Kubernetes. [Online]. Available: https://kubernetes.io/docs/

[24] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[25] Kubernetes, "Scheduling framework," https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, accessed: 2024-08-13.

[26] Kubernetes Scheduler Extender. Accessed: 2024-08-09. [Online]. Available: https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md

[27] Kubernetes scheduler plugins. Accessed: 2024-10-09. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins

[28] The Linux Foundation, "Customizing k8s scheduler," https://cncf.io/blog/2022/04/19/customizing-k8s-scheduler/, accessed: 2024-11-09.

[29] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[32] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[33] M. Gusev, S. Ristov, M. Simjanoska, and G. Velkoski, "Cpu utilization while scaling resources in the cloud," *Cloud Computing*, pp. 131–137, 2013.

[34] Knative platform. [Online]. Available: https://knative.dev/docs/

[35] Kepler metrics. [Online]. Available: https://sustainable-computing.io/design/metrics/

[36] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. Pmlr, 2014, pp. 387–395.

[37] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[38] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[39] M. Adeppady, A. Conte, H. Karl, P. Giaccone, and C. F. Chiasserini, "Energy-aware Provisioning of Microservices for Serverless Edge Computing," in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, 2023, pp. 3070–3075.

[40] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and M. B. Chhetri, "faasHouse: Sustainable Serverless Edge Computing through Energy-aware Resource Scheduling," *IEEE Transactions on Services Computing*, 2024.

[41] S. Arys, R. Carlier, and E. Rivière, "Energy-Aware Scheduling of a Serverless Workload in an ISA-Heterogeneous Cluster," in *Proceedings of the 10th International Workshop on Serverless Computing*, ser. WoSC10 '24. Association for Computing Machinery, 2024, p. 25–30.

[42] S. H. Rastegar, H. Shafiei, and A. Khonsari, "EneX: An Energy-Aware Execution Scheduler for Serverless Computing," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 2, pp. 2342–2353, 2024.

[43] R. Yang, X. Ouyang, Y. Chen, P. Townend, and J. Xu, "Intelligent Resource Scheduling at Scale: A Machine Learning Perspective," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 132–141.

[44] E. Hormozi, H. Hormozi, M. K. Akbari, and M. S. Javan, "Using of Machine Learning into Cloud Environment (A Survey): Managing and Scheduling of Resources in Cloud Systems," in *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 2012, pp. 363–368.

[45] J. Rothman and J. Chamanara, "An RL-Based Model for Optimized Kubernetes Scheduling," in *2023 IEEE 31st International Conference on Network Protocols (ICNP)*. IEEE, 2023, pp. 1–6.

[46] R. Douhara, Y.-F. Hsu, T. Yoshihisa, K. Matsuda, and M. Matsuoka, "Kubernetes-based workload allocation optimizer for minimizing power consumption of computing system with neural network," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2020, pp. 1269–1275.

[47] R. Chiorescu and K. Djemame, "Scheduling energy-aware multi-function serverless workloads in openfaas," in *Economics of Grids, Clouds, Systems, and Services*. Cham: Springer Nature Switzerland, 2025, pp. 137–149.