






# *POSEIDON*: Efficient Function Placement at the Edge using Deep Reinforcement Learning

Prakhar Jain<sup>1</sup>, Prakhar Singhal<sup>1†</sup>, Divyansh Pandey<sup>1†</sup>, Giovanni Quattrocchi<sup>2</sup>, and Karthik Vaidhyanathan<sup>1</sup>

<sup>1</sup> Software Engineering Research Center, International Institute of Information Technology  
Hyderabad, India

{prakhar.jain, prakhar.singhal}@research.iiit.ac.in,  
divyansh.pandey@students.iiit.ac.in, karthik.vaidhyanathan@iiit.ac.in

<sup>2</sup> Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria  
Milan, Italy  
giovanni.quattrocchi@polimi.it

**Abstract.** Edge computing allows for reduced latency and operational costs compared to centralized cloud systems. In this context, serverless functions are emerging as a lightweight and effective paradigm for managing computational tasks on edge infrastructures. However, the placement of such functions in constrained edge nodes remains an open challenge. On one hand, it is key to minimize network delays and optimize resource consumption; on the other hand, decisions must be made in a timely manner due to the highly dynamic nature of edge environments. In this paper, we propose *POSEIDON*, a solution based on Deep Reinforcement Learning for the efficient placement of functions at the edge. *POSEIDON* leverages Proximal Policy Optimization (PPO) to place functions across a distributed network of nodes under highly dynamic workloads. A comprehensive empirical evaluation demonstrates that *POSEIDON* significantly reduces execution time, network delay, and resource consumption compared to state-of-the-art methods.

**Keywords:** Edge Computing · Serverless · Function Placement · Deep Reinforcement Learning

## 1 Introduction

Edge computing has emerged as a promising solution to address the limitations of centralized cloud systems, particularly in terms of reducing latency and operational costs. This paradigm shift enables computational tasks to be processed closer to the data source, thereby enhancing the performance and efficiency of applications [14]. In this decentralized framework, edge nodes, by their nature, are often constrained in terms of resources such as processing power, memory,

---

<sup>†</sup> These authors contributed equally to this work.

and storage. Additionally, the workload on these nodes can be highly fluctuating, with varying demands depending on user activities and their dynamic geographical location. This inherent variability necessitates a flexible and efficient approach to managing applications running on edge nodes [26].

Recently, the serverless paradigm has emerged as a suitable solution for managing applications in edge computing infrastructures [17]. Serverless allows developers to deploy applications as a collection of discrete, self-contained functions that are designed to be lightweight and stateless [24]. In the context of edge computing, such functions can be quickly moved across nodes to adapt to the mobility of users and the shifting demands of applications.

Despite the advantages, dynamically placing serverless functions in edge nodes presents significant challenges [19]. Ideally, functions should be placed as close to users as possible to minimize network delays and optimize resource consumption. However, resource-constrained edge cannot always host all necessary functions and the mobility of users and the heterogeneity of functions in terms of CPU and memory requirements further complicate the problem. Moreover, such a dynamic environment requires timely decisions to cope with fluctuating workloads [3]. In the literature, most of the work exploits combinatorial optimization techniques, such as Integer Programming formulations, to solve the placement problem effectively [2, 7, 8]. While these methods are capable of producing optimal solutions, they are often complex and time-consuming, resulting in slow solution generation. Some approaches have proposed custom heuristics as an alternative [11], but these methods have been demonstrated to produce significantly lower quality placements compared to optimization-based approaches [3].

In this context, we introduce *POSEIDON*<sup>1</sup>, a novel solution that utilizes Deep Reinforcement Learning [9] (DRL) to optimize the placement of serverless functions at the edge. *POSEIDON* specifically leverages Proximal Policy Optimization [21] (PPO) to distribute functions across a network of nodes, effectively managing highly dynamic workloads. After determining the placement, *POSEIDON* uses a simplified Mixed Integer Linear Programming (MILP) problem to optimize traffic routing across the different function instances. The proposed method aims to reduce network delays, improve resource consumption, and produce timely solutions compared to existing state-of-the-art techniques.

The high-dimensional nature of the placement problem and continuous state space render classical RL algorithms impractical. Traditional RL typically relies on discrete state-action spaces, which are unsuitable for complex scenarios such as edge computing. DRL overcomes this limitation by using neural networks to approximate complex mappings from states to actions or action probabilities, enabling efficient and scalable learning in high-dimensional state spaces. Moreover, being known for its ability to capture intricate interactions between states and actions, DRL is well-suited to handle real-world environments that are complex and dynamic, requiring flexible and adaptive learning methods [9, 13].

---

<sup>1</sup> <https://github.com/sa4s-serc/poseidon>

We evaluated *POSEIDON* through a comprehensive comparison with state-of-the-art solutions. Our extensive empirical evaluation demonstrated that *POSEIDON* is almost 16 times faster than the state-of-the-art with respect to decision time with almost comparable cost and delay to the state-of-the-art.

The rest of the paper is organized as follows. Section 2 presents the problem and introduces our solution. Section 3 details the DRL solution and the MILP formulation. Section 4 presents the empirical evaluation of *POSEIDON* and the comparison against the state-of-the-art. Section 5 describes some relevant work and concludes with Section 6.

## 2 Problem and Solution Overview

In *POSEIDON*, an edge topology is defined as a graph where  $N$  is the set of nodes and the edges are the links between them. Each pair of nodes  $i$  and  $j$  is characterized by  $\delta_{ij}$ , representing the network delay between them.  $F$  is the set of functions that could be deployed on the edge topology. We assume that users can connect to any of the nodes in  $N$  based on their geographical proximity to the closest one. Thus, for each function  $f$  and node  $i$ , the incoming workload for a function  $f$  to node  $i$  is defined as  $w_{f,i}$ , representing the number of requests for  $f$  arriving at node  $i$ . Since each node is resource-constrained and cannot host all functions, we assume that each node can route the requests to any other (nearby) node  $j$ . Thus, the problem *POSEIDON* tackles is twofold:

1. Deciding whether an instance of function  $f$  should be placed on node  $i$  (*placement*);
2. Deciding how the workload incoming to any node  $i$  for function  $f$  should be routed to any other node  $j$  (*routing policies*).

### 2.1 Solution overview

To address these problems, *POSEIDON* leverages DRL for placement and a MILP formulation for routing policies. The goal of *POSEIDON* is to minimize both (i) the overall network delay, i.e., the latency while serving function requests, and (ii) the cost of running the function instances. The former goal focuses on placing functions as close to users as possible to minimize routing delays. The latter goal aims to use the minimum number of nodes to serve all the workload with minimal overhead. *POSEIDON* ensures a balance between these conflicting objectives using a user-defined trade-off. *POSEIDON* works in three phases and its architecture is shown in Figure 1.

The *placement phase*, detailed in Section 3.1, is dedicated to computing the function placement. To achieve this, *POSEIDON* organizes the functions  $F$  into a queue  $q_F$ , prioritized by their specific criteria such as frequency of requests or resource requirements. Then, a DRL agent considers each function one at a time, starting with the highest priority. For each function, the agent computes a placement vector, a boolean vector  $\mathbf{c}_i^f$ , which indicates whether a function

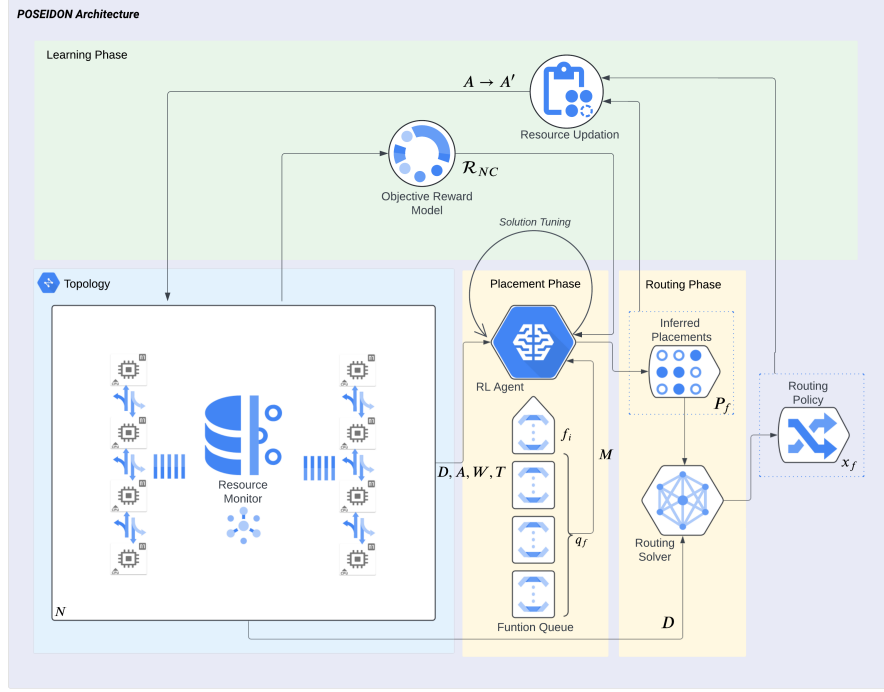


Fig. 1: POSEIDON architecture.

$f$  should be placed on node  $i$ . The agent leverages comprehensive information from the topology, including inter-node delays, available hardware resources, and function parameters such as memory requirements. It also considers the workload of the function instance, reflecting the traffic of function requests at each node. This information is used to infer an optimal function placement that satisfies our objectives.

In the *routing phase*, the function placement determined by the agent is used to compute the routing policies, as detailed in Section 3.2. In this phase, every node  $i$  could host an active function instance and/or route a portion of the requests to another node  $j$  with a previously placed  $f$  instance. This is done by computing matrix  $\mathbf{x}_{i,j}^f$ , where each value is a real number between 0 and 1 that represents the fraction of the workload incoming to node  $i$  for function  $f$  that should be routed to node  $j$ . To compute the routing policies, POSEIDON utilizes the generated function placements to formulate a MILP problem. The constraints of this problem ensure that all function requests are redirected to nodes with a running instance of the function. The objective is to minimize the network delay of function requests and the cost of running the functions, thereby aligning with the overall solution objective.

In the *learning phase*, the state of the topology is updated, and a reward  $\mathcal{R}$  is calculated, as detailed in Section 3.3. This reward mechanism ensures that the DRL agent learns to place the functions in a manner that minimizes both the total network delay and the operational costs of the function instances.

### 3 POSEIDON

Table 1: Inputs, Outputs and Reward Variables

Component	Symbol	Description
<b>Topology Data</b>		
Inter-Node Delay	$\delta_{i,j}$	Network delay between node $i$ and node $j$
Available node memory	$m_i$	Available memory on node $i$
Available node cores	$k_i$	Available CPU cores on node $i$
<b>Function Data</b>		
Function Workload	$w_{f,i}$	Workload for function $f$ on node $i$
Function memory	$m_f^R$	Memory required by function $f$
Function cores	$k_{f,i}^R$	Average CPU cores required by instances function $f$ on node $i$
<b>Monitored Data</b>		
Total Delay	$T$	Total network delay
Total Cost	$C$	Total cost of running placed functions
<b>Output Data</b>		
Placement vector	$c_{f,i}$	Boolean decision variable representing whether function $f$ is placed on node $i$ .
Routing policy	$x_{f,i,j}$	Real decision variable representing the fraction of $f$ workload incoming into node $i$ to be routed to node $j$

This section details the three phases of *POSEIDON*. To facilitate the read, we included the most important variables used in our formulation in Table 1.

#### 3.1 Function Placement

*POSEIDON* uses five vectors to represent the state of the topology,  $D$ ,  $A$ ,  $W$ ,  $M$ ,  $T$ .  $D$  is the delay vector where each element is the inter-node delay  $\delta_{i,j}$ ,  $A$  is the available resources,  $W$  is the workload of the current function  $f$ ,  $M$  encodes the parameters of the current function and  $T$  denotes the cumulative delay of the topology. The delay vector  $D$  can be represented as:

$$D = [\delta_{1,1} \dots \delta_{1,n} \delta_{2,1} \dots \delta_{2,n} \dots \delta_{n,1} \dots \delta_{n,n}]$$

with  $\delta_{i,i} = 0$  (i.e., local communications do not have delays) and  $\delta_{i,j} = \delta_{j,i}$  (i.e., delay is symmetric) for any node  $i$  and  $j$ .

The available resources in the topology  $A$ , consist of the available memory  $m_i$  and available CPU cores  $k_i$  of each node  $i$ . This variable captures the available hardware resources at each node and is updated after placing each function instance to account for the resources consumed by the placed functions. Formally,

$$A = [k_1 \ m_1 \ k_2 \ m_2 \ \dots \ k_n \ m_n]$$

The workload of the current function  $f$ , which is denoted by  $W$ , is composed of the function workload at each node  $i$  denoted by  $w_{f,i}$ . Such data helps determine whether a function needs to be placed on a specific node based on the amount of function requests being received by it. The function parameters are passed into the state vector in the form of a vector  $M$  consisting of  $m_f^R, m_\mu^R, m_\sigma^R$  where the latter two represent the mean and standard deviation of the resource requirements required by the remaining functions in  $q_F$ . These guide the placement decisions while keeping track of the resource requirements of the remaining functions.

$$M = [m_f^R \ m_\mu^R \ m_\sigma^R]$$

The total network delay  $T$  of the topology initially set to 0 is continuously monitored and updated after placing each function  $f$  as follows:

$$T = T + \sum_i^N \sum_j^N x_{f,i,j} * w_{f,i} * \delta_{i,j}$$

where  $x_{f,i,j}$  is the routing variable for routing requests of function  $f$  from node  $i$  to  $j$  and is computed after the second phase, as described in Section 3.2

Feeding the cumulative total delay to the agent helps it learn to minimize the delay effectively based on the existing network delay of the system.

The DRL agent implemented in *POSEIDON* system employs a policy gradient method known as Proximal Policy Optimization (PPO) to learn an optimal policy  $\pi$ . This policy maps a given state  $s \in \mathcal{S}$  of the topology (i.e., the execution environment) to an action  $c \in \mathcal{A}$ , where  $\mathcal{A} \in \{0, 1\}^N$  represents the action space of the agent. This action space comprises of a set of boolean values for each node, indicating whether a function instance should be placed on the corresponding node or not. The state  $s$  exists within a continuous state space and encapsulates environmental information that affect the placement of the function and is formally defined as a feature vector as follows:

$$s = [D \ A \ W \ M \ T]^\top$$

Since  $s$  is high dimensional and in a continuous state space it is difficult to model the placement problem with RL methods such as Q-Learning which requires tabulation and hence impractical to store and update the table, the need for DRL arises. The agent consists of a neural network that takes  $\mathcal{S}$  as input

and estimates a policy  $\pi^*$  which infers actions  $\mathcal{A}^*$ . These actions are used to compute the reward  $\mathcal{R}$  for the agent after a suitable routing policy is computed, as mentioned in Section 3.3.

### 3.2 Routing policies

Function placement is not sufficient to minimize the total network delay of the system because a node with the running instance of the function may not be able to handle the heavy workload expected in edges networks. In such cases, the need to route a certain fraction of function requests to other available nodes arises which would prevent overloading the node with user requests. To handle the routing of function requests, the objectives are formulated as a Mixed Integer Programming problem where the constraints are defined as follows:

$$P_f = \{i \mid c_{f,i} = 1, i \in N\} \quad \forall f \in F \quad (1)$$

where  $P_f$  is the set of chosen nodes for placing a function. Thus, the following constraint ensures that all the  $f$  workload from node  $i$  are routed only to the nodes that belong to  $P_f$ .

$$\sum_j^{P_f} x_{f,i,j} = 1 \quad \forall i \in N, \forall f \in F \quad (2)$$

On the contrary, the following equation states that no fraction of  $f$  workload should be routed from any node  $i$  to a node  $j$  that does not belong to  $P_f$ .

$$x_{f,i,j} = 0 \quad \forall i \in N, \forall f \in F, \forall j \notin P_f \quad (3)$$

For each node  $j$  that host a function  $f$ , the required CPU cores to handle the total workload forwarded to the node should be lower than the available cores on that node. This is given by:

$$\sum_i^N x_{f,i,j} * w_{f,i} * k_{f,j}^R \leq k_j \quad \forall f \in F, \forall j \in P_f \quad (4)$$

The objective of the problem is to minimize the network delay of the placed function. The network delay  $T_f$  for a specific function  $f$  is defined as:

$$T_f = \sum_i^N \sum_j^N x_{f,i,j} * w_{f,i} * \delta_{i,j} \quad (5)$$

### 3.3 State Update and Solution Tuning

Based on the updated state of the community, the reward  $\mathcal{R}$  is calculated to facilitate the parameter updates and solution tuning (Algorithm 2). Solution tuning involves training on the current state  $s$  of the system to improve the

---

**Algorithm 1** Update state after placing function  $f$ 

---

**Require:**  $A, M, q_F, P_f$   
1: **for** each  $i \in P_f$  **do**  
2:    $m_i \leftarrow m_i - m_f^R$   
3:    $k_i \leftarrow k_i - \sum_j^N w_{f,j} * x_{f,j,i} * k_{f,i}^R$   
4: **end for**  
5:  $pop(q_F)$   
6:  $f = \text{peek}(q_F)$

---



---

**Algorithm 2** Episodic reward calculation after placing function  $f$ 

---

1: **Input:**  $T, C, T_{\min}$  (minimum observed total network delay),  $T_{\max}$  (maximum observed total network delay),  $C_{\min}$  (minimum observed cost),  $C_{\max}$  (maximum observed cost)  
2: **Output:**  $T'$  (normalized total network delay),  $C'$  (normalized cost),  $\mathcal{R}$  (reward)  
3: Calculate  $T'$  and  $C'$  using:  

$$T' = 2 \left( \frac{T - T_{\min}}{T_{\max} - T_{\min}} \right) - 1 \quad C' = 2 \left( \frac{C - C_{\min}}{C_{\max} - C_{\min}} \right) - 1$$
  
4: **if**  $\text{size}(P_f) = 0$  or  $m_i < 0$  or  $k_i < 0$  or  $\text{RoutingSolver.STATUS} == \text{INFEASIBLE}$  **then**  
5:    $\mathcal{R} = \mathcal{R}_{\text{penalty}}$   
6: **else**  
7:    $\mathcal{R} = \mathcal{R}_{NC}$   
8: **end if**  
9: Update  $T_{\min}, T_{\max}, C_{\min}$ , and  $C_{\max}$ .  
10:  $\text{UpdateState}(A, M, f, P_f)$   
11: **if**  $q_F$  is empty **then**  
12:   **Return Reward and end episode**  
13: **else**  
14:   **Return Reward and repeat the process.**  
15: **end if**

---

DRL agent . The reward model for the environment is designed to ensure that the agent comprehends the objectives of the problem and refines the policy  $\pi^*$ . The agent receives different types of rewards based on its decisions. The network delay and the system's operational cost are incorporated into a reward term, formulated to be minimized by the agent to improve its performance. Given the conflicting nature of minimizing network delay and system cost, a user-defined trade-off parameter  $\alpha$  is introduced as coefficient of trade off with value in range  $(0, 1)$  where increasing  $\alpha$  corresponds to increasing weightage of cost minimization objective and decreasing  $\alpha$  corresponds to increasing weightage of delay minimization objective. Mathematically, the reward term is defined as:

$$\mathcal{R}_{NC} = -(\alpha C' + (1 - \alpha)T') \quad (6)$$



where  $T'$  and  $C'$  are normalized values of the total network delay and the system's cost, respectively. Normalization is employed to linearly scale these terms between  $[-1,1]$ , ensuring equal upper and lower bounds for both objectives. This solves the scaling problem for different units, thereby maintaining a balanced consideration of both network delay and cost (line 3 of Algorithm 2). The maximum processing times, denoted as  $T_{max}$  is computed based on the edge-case scenario. This scenario assumes a complete cyclic routing, wherein each request must traverse through all nodes before being processed, whereas  $T_{min}$  is set to zero reflecting case where all requests can directly be served without need of routing any to other nodes.

$\mathcal{R}_{NC}$  is not sufficient to guide the agent for the objectives of *POSEIDON*. The agent may attempt to exploit the system by not placing the functions on any nodes, or it may make invalid decisions, such as placing functions on nodes with insufficient compute power or generating placements that result in infeasible mixed-integer programming (MIP) solutions during the routing phase. To discourage such undesirable behavior, the agent receives a negative reward for each invalid placement decision made by the agent which involves violating compute resource, violating memory resources and violating routing constraint represented by  $R_{Penalty}$  (line 5 of Algorithm 2), strictly guiding it away from these solutions.

$$R_{Penalty} = N(\Omega_i)$$

where  $\Omega_i$  denotes an invalid placement decision.

## 4 Evaluation

The objective of the experiments is to evaluate the effectiveness and efficiency of our approach by answering the following questions:

**RQ1.** How does *POSEIDON* compare to state-of-the-art solutions in terms of delay and cost?

**RQ2.** How does *POSEIDON* perform with respect to decision time compared to state-of-the-art solutions?

**RQ3.** How does solution tuning in *POSEIDON* mitigate invalid placements?

### 4.1 Experimental Setup

**Execution Environment:** We implemented a simulated edge environment using the Gymnasium by providing the specifications of the nodes and functions as inputs to the simulation. Function requests were sampled using the Cabspotting [15] dataset, wherein the node delays between the nodes were predefined. The DRL agent was configured to learn within the Gymnasium environment<sup>2</sup>, employing the Proximal Policy Optimization (PPO) algorithm, as implemented by the Stable-Baselines3 library<sup>3</sup>. The routing solver was integrated into the system

<sup>2</sup> <https://gymnasium.farama.org>

<sup>3</sup> <https://github.com/hill-a/stable-baselines>

using the Linear Solver provided by Google’s OR-Tools. The *POSEIDON* was trained for 50 workload distributions at different timesteps and then evaluated on 150 different workload samples. Our simulation was run on an Ubuntu 23.04 machine with 16 GB RAM, powered by a 4.6 GHz 12th Gen Intel i7 processor, and an Nvidia RTX 3060 GPU with 6 GB VRAM.

**Experiment Candidates:** We evaluated *POSEIDON*<sup>4</sup> by performing four simulations this was done by placing different number of functions on 5 nodes equipped with 50, 50, 50, 25, 100 cores and memory capacities of 100, 100, 200, 50, 500 GB. The first two simulations, with  $\alpha = 0$  and  $\alpha = 0.5$ , involved placing a *small payload* of 4 functions with memory requirements of 50, 10, 10, 10 GB whereas the third and the fourth simulation with  $\alpha = 0$  and  $\alpha = 0.5$  respectively involved placing a *large payload* of 10 functions with memory requirements of 10 GB each on the same set of nodes as described earlier. For each simulation, the agent was trained using 50 workloads for 2000 timesteps, with each timestep lasting 100 seconds. The data used for training came from the *Cabspotting* [15] dataset. The purpose of these simulations was to assess *POSEIDON*’s efficiency in not only minimizing the cost and delay but the decision time as well.

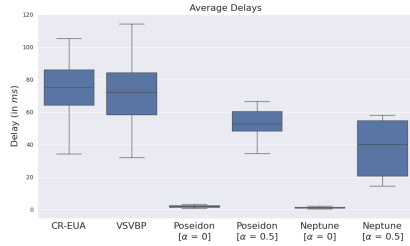
1. *CR-EUA* : Criticality-Awareness Edge User Allocation [10]: an allocation strategy tailored for safety-critical, low-latency applications where meeting strict performance and reliability requirements is paramount. This approach aims to maximize the number of requests processed at the highest level of criticality by intelligently assigning edge resources to the most critical tasks. By prioritizing requests based on their criticality levels, CR-EUA ensures that applications requiring immediate attention receive the necessary computational resources to function optimally.

2. *VSVBP* (Variable Sized Vector Bin) [6]: a placement and routing approach designed to optimize resource utilization in edge computing networks. It works by maximizing the number of allocated service requests while minimizing the number of active edge nodes, thereby reducing operational costs and energy consumption. The method ensures that the response times of deployed services stay within acceptable limits by efficiently distributing workloads across the available resources. VSVBP models the resource allocation problem as a variable-sized bin packing issue, which makes it ideal for scenarios where both performance and cost-effectiveness are critical

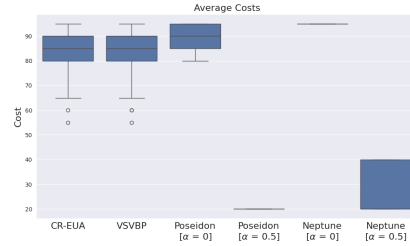
3. *NEPTUNE* [2, 3]: a solution designed for optimal placement of serverless functions on edge nodes using MIP with the goal of minimizing both network latency and resource utilization. NEPTUNE optimizes the placement of functions by considering factors like network delay and minimizing the number of active nodes to reduce operational costs. After the initial placement, it uses a second optimization step to reduce service disruptions. Further, NEPTUNE also generates routing policies that direct traffic to the appropriate nodes, balancing the

---

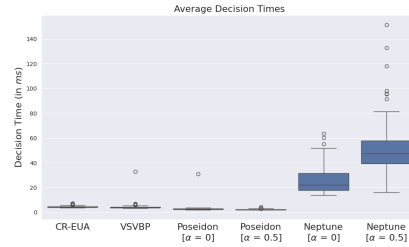
<sup>4</sup> <https://github.com/sa4s-serc/poseidon>



(a) Box plot depicting the delay while using each of the different approaches



(b) Box plot depicting the cost while using each of the different approaches



(c) Box plot depicting the average decision time while using each of the different approaches

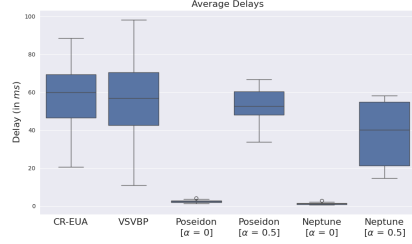
Fig. 2: Effectiveness of the approaches with respect to various metrics for the *small payload*

workload and minimizing inter-node delays, while ensuring functions are only terminated after they complete their current tasks.

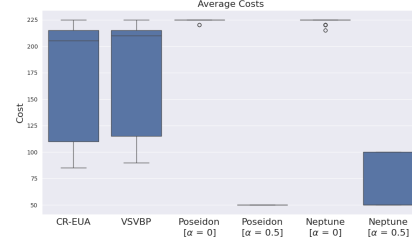
**Evaluation Metrics** To measure the effectiveness and efficiency of the approach, we use three different metrics: *i)* Total Delay: this represents the sum of the delay for each of the function requests in the topology, *ii)* Cost: this accounts for the total cost of running the function instances on the nodes, *iii)* Decision time: this metric evaluates how quickly the placement and routing of functions are determined, reflecting the computational efficiency of the approach.

#### 4.2 Delay and cost analysis (RQ1)

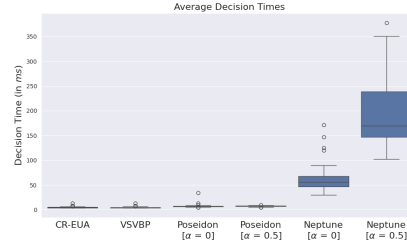
To compare the effectiveness of *POSEIDON* for cost and delay, we evaluated *POSEIDON* with four different simulations as mentioned in Section 4.1. We can see from the Tables 2 and 3 and Figures 2 and 3 that when  $\alpha = 0$  in case of both *large* and *small payload* the average delay is very low but the average cost is high for both *POSEIDON* and *NEPTUNE*. This is expected because for  $\alpha = 0$  both *POSEIDON* and *NEPTUNE* try to minimize the delay and the cost is not considered a critical factor leading to the high average cost. Also both *POSEI-*



(a) Box plot depicting the delay while using each of the different approaches



(b) Box plot depicting the cost while using each of the different approaches



(c) Box plot depicting the average decision time while using each of the different approaches

Fig. 3: Effectiveness of the approaches with respect to various metrics for the *large payload*

*DON* and *NEPTUNE* outperform the other approaches when it comes to delay for similar average costs. *VSVBP* and *CR-EUA* exhibit higher delays because they prioritize maximizing resource utilization and processing the most critical tasks. This focus on handling critical requests often leads to resource overloading, increasing processing times. On increasing  $\alpha$  from 0 to 0.5, averaging over *small* and *large* payloads, the average costs are much lower, dropping by 77.74 % for *POSEIDON* and 71.52% for *NEPTUNE*, the average delays increases by 24.9 times for *POSEIDON* and 33.69 times for *NEPTUNE*, as both approaches try to find a balance between the delay and the cost. Albeit overall, *POSEIDON* has a lower cost and higher delay than *NEPTUNE*. Even though *POSEIDON* is not as optimal as *NEPTUNE* but is comparable to *NEPTUNE*. *CR-EUA* and *VSVBP* exhibit higher costs than *POSEIDON* because they prioritize processing the maximum number of requests, which requires allocating additional resources. Moreover, the stability of *POSEIDON* can be attributed to its use of PPO, which is known for its robustness and ability to produce stable policies. The training of the DRL agent ensures that that *POSEIDON* learns to handle varying conditions in a controlled manner.

Table 2: Comparison of *POSEIDON* with Other Approaches for *small payload*  $\alpha = 0$  and  $\alpha = 0.5$ 

Metric	Poseidon		Neptune		CR-EUA	VSVBP
	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 0$	$\alpha = 0.5$		
Average Delay (in <i>ms</i> )	1.9420	53.1460	1.1060	38.7830	74.4574	71.6157
Average Cost	90.7000	20.2000	95.9500	26.4000	85.8500	85.1000
Average Decision Time (in <i>ms</i> )	3.1	2.4	26	52.1	4.5	4.5

Table 3: Comparison of *POSEIDON* with Other Approaches for *large payload*  $\alpha = 0$  and  $\alpha = 0.5$ 

Metric	Poseidon		Neptune		CR-EUA	VSVBP
	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 0$	$\alpha = 0.5$		
Average Delay (in <i>ms</i> )	2.3156	51.8161	1.1744	37.6083	57.0778	54.1870
Average Cost	227.3333	50.5556	226.5000	66.6667	167.7222	178.1667
Average Decision Time (in <i>ms</i> )	7.3	7.3	61.7	190.6	4.7	4.6

### 4.3 Decision time analysis (RQ2)

To evaluate the efficiency of *POSEIDON* with respect to decision time compared to other approaches, we ran four different simulations as mentioned in 4.1 we can see from the Tables 2 and 3 and Figures 2c and 3c *POSEIDON* demonstrates consistent decision times for  $\alpha = 0$  and  $\alpha = 0.5$  for both the *small payload* and *large payload* whereas for NEPTUNE the decision time is higher. This can be attributed to the fact that *POSEIDON* divides the solution into two parts: i) the DRL agent solves the placement problem and has a very small number of parameters, which enables faster decision-making with regards to placements, and ii) the routing solver which solves the routing problem uses MILP but has fewer constraints than NEPTUNE making *POSEIDON* 16.43 times faster. This also further highlights the scalability challenges incurred by NEPTUNE due to the computational complexity of solving the MILP problem with too many constraints. The other approaches, CR-EUA and VSVBP show comparable decision times to *POSEIDON* but mostly give sub-optimal solutions with respect to cost and delay.

### 4.4 Impact of solution tuning (RQ3)

Figure 4 shows the cumulative number of invalid placements performed by the agent against the iterations of the solution tuning cycle. The graph shows a converging curve, which can be correlated with the agent’s overall improvement over the iterations as it learns to avoid invalid placements, which can be attributed to  $R_{Penalty}$  (refer Section 3). We observe an average decrease of 49.13% in the aggregated number of invalid placements per workload’s tuning iteration, indicating a strong positive learning trend.

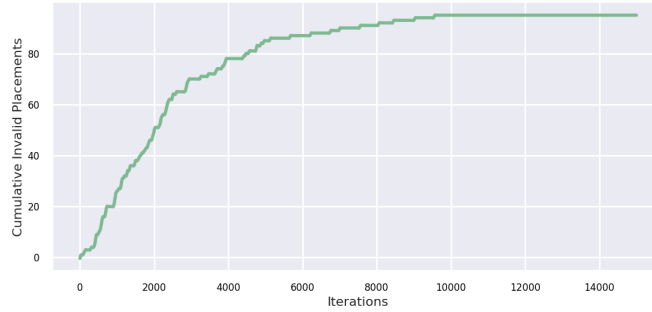


Fig. 4: Cumulative count of invalid placements versus iterations of solution tuning for *small payload* ( $\alpha = 0$ )

#### 4.5 Threats to Validity

Threats to *Construct Validity* concern the use of controlled experimental setup and simulations. To this end, we ensured that we used workloads and configurations as close to the real-world scenario as possible. Specifically, we utilized the *Cabspotting* dataset [15] to train and test *POSEIDON*. However, it is important to note that while our simulations provide a controlled environment for evaluation, further experiments in real-world are necessary to fully assess the practical applicability and effectiveness of our findings.

Threats to *Internal Validity* concern the use of a static number of nodes for placements. In practice, the DRL agent needs to be retrained if there is a change in the topology, such as the addition or removal of a node. Although we can train the DRL agent to minimize the occurrence of invalid or infeasible placements, the nature of machine learning means that we cannot guarantee that the agent will always produce valid placements. Therefore, it is essential to have an external system in place to verify the placement decisions made by the DRL agent, ensuring their correctness and feasibility.

Threats to *External Validity* of our approach concern the generalizability and scalability of our approach. Although our approach has been applied to two different scenarios with 4 functions and 10 functions, respectively, the techniques used in the approach are scalable to a larger number of functions. This is further validated by the results as demonstrated in Section 4.3. As regards to the generalizability, the approach can be integrated to any MEC system as long as it provides with mechanisms to monitor the function parameters as well as to perform routing and placement.

## 5 Related Work

Deploying applications on edge infrastructures has increasingly become the preferred method to meet the rising demand for low-latency applications [23]. Thus,

the placement of such applications and their request routing in edge systems has become a primary research focus [19] since existing solutions dedicated to cloud-computing often neglect the unique challenges of the edge context [5] such as managing the geographical distribution of computing nodes, maintaining low network latency, and coping with resource constrained nodes [17]. Numerous studies have tackled the challenges of placement and routing in edge computing [18, 20]. A common approach involves framing the problem of service placement and workload routing as a *Integer Programming* problem [7, 8].

For example, NEPTUNE [2] utilizes a MILP formulation to place serverless functions on edge nodes. Given computed placement and routing policies, NEPTUNE optimizes resources using vertical scaling controllers based on control-theory [4, 16]. Compared to NEPTUNE, *POSEIDON* shares similar objectives but employs Reinforcement Learning for computing placements. This allows for timely computation of new placements and better adaptation to dynamic and fluctuating environments, such as edge computing. We view our approach as complementary to NEPTUNE: *POSEIDON* can be embedded within NEPTUNE communities to determine placements, while a simplified version of NEPTUNE can then be used to compute routing policies based on these placements.

Unlike *POSEIDON*, which reduces latency by placing applications closer to users, Ma et al. [12] focus on maximizing edge node utilization using MILP without considering network delays as we do. Liu et al. [11] address network delays by prioritizing requests based on criticality and response times, although the approach doesn't explicitly aim to minimize delays. *POSEIDON* does not explicitly consider the criticality of functions. However, one can prioritize critical application by affecting the ordering mechanism of functions employed in our RL-based approach. Finally, Tong et al. [22] utilize Mixed Nonlinear Integer Programming to maximize served requests in a hierarchical MEC network. The main benefit of *POSEIDON* compared to ones based on combinatorial optimization is to provide solutions in an efficient and timely manner which allow to better cope with edge nomadic users and highly-fluctuating workloads.

Raza et al. [1] present COSE, a framework that uses Bayesian Optimization to find the optimal resource configuration and placement for serverless applications, minimizing cost while meeting performance objectives. COSE provides an efficient, non-combinatorial, solution to the problem; however, compared to *POSEIDON*, it focuses on cloud architectures and does not consider the intrinsic characteristics of edge topologies.

Xu et al [25] propose an adaptive function placement framework utilizing a Markov Decision Process to optimize serverless computing performance across terminal devices, edge nodes, and cloud data centers. The framework supports adaptation in real-time to allocate functions dynamically, aiming to minimize execution costs while maintaining performance satisfaction. Compared to the *POSEIDON*, the approach does not consider memory requirements, network delay among edge nodes, and routing times, which are central for edge computing. Moreover, while the approach aim to optimize the utilization of the available devices, *POSEIDON* also minimizes the overall network delay.

## 6 Conclusions

In this paper, we introduced *POSEIDON*, an innovative method that integrates Deep Reinforcement Learning (DRL) with traditional optimization techniques, specifically Mixed Integer Linear Programming (MILP), to tackle the challenge of serverless function placement in edge infrastructures. Our evaluation shows that *POSEIDON* makes decisions that are near-optimal in terms of cost and delay, while also achieving low decision-making times. For future work, we aim to leverage workload predictions to proactively place function instances and foresee potential resource saturation.

## References

1. Ali, R., Nabeel, A., Isahagian, V., Ibrahim, M., Lei, H.: Configuration and placement of serverless applications using statistical learning. *IEEE Transaction On Network and Service Management* pp. 1065 – 1077 (2023)
2. Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L.: NEPTUNE: network- and gpu-aware management of serverless functions at the edge. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. pp. 144–155. ACM (2022)
3. Baresi, L., Hu, D.Y.X., Quattrocchi, G., Terracciano, L.: Neptune: A comprehensive framework for managing serverless functions at the edge. *ACM Trans. Auton. Adapt. Syst.* **19**(1) (feb 2024)
4. Baresi, L., Quattrocchi, G.: Towards vertically scalable spark applications. In: *Proceedings of the Parallel Processing Workshops*. vol. 11339, pp. 106–118. Springer (2018)
5. Bellendorf, J., Mann, Z.Á.: Classification of optimization problems in fog computing. *Elsevier Future Generation Computer Systems* **107**, 158–176 (2020)
6. Lai, P., He, Q., Abdelrazek, M., Chen, F., Hosking, J., Grundy, J., Yang, Y.: Optimal edge user allocation in edge computing with variable sized vector bin packing. In: *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*. pp. 230–245. Springer (2018)
7. Lai, P., He, Q., Abdelrazek, M., Chen, F., Hosking, J.G., Grundy, J.C., Yang, Y.: Optimal edge user allocation in edge computing with variable sized vector bin packing. In: *Proceedings of the International Conference on Service-Oriented Computing*. vol. 11236, pp. 230–245. Springer (2018)
8. Lai, P., He, Q., Grundy, J., Chen, F., Abdelrazek, M., Hosking, J.G., Yang, Y.: Cost-effective app user allocation in an edge computing environment. *IEEE Transactions on Cloud Computing* **10**(3), 1701–1713 (2022)
9. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning (2019)
10. Liu, E., Zheng, L., He, Q., Xu, B., Zhang, G.: Criticality-awareness edge user allocation for public safety. *IEEE Transactions on Services Computing* **16**(1), 221–234 (2021)
11. Liu, E., Zheng, L., He, Q., Xu, B., Zhang, G.: Criticality-awareness edge user allocation for public safety. *IEEE Transactions on Service Computing* **16**(1), 221–234 (2023)



12. Ma, Z., Zhang, S., Chen, Z., Han, T., Qian, Z., Xiao, M., Chen, N., Wu, J., Lu, S.: Towards revenue-driven multi-user online task offloading in edge computing. *IEEE Transactions on Parallel and Distributed Systems* **33**(5), 1185–1198 (2022)
13. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. *nature* **518**(7540), 529–533 (2015)
14. Pham, Q., Fang, F., Ha, V.N., Piran, M.J., Le, M., Le, L.B., Hwang, W., Ding, Z.: A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art. *IEEE Access* **8**, 116974–117017 (2020)
15. Piorkowski, M., Sarafijanovic-Djukic, N., Grossglauser, M.: A parsimonious model of mobile partitioned networks with clustering. In: *Proceedings of the International Communication Systems and Networks and Workshops*. pp. 1–10. IEEE (2009)
16. Quattrocchi, G., Incerto, E., Pinciroli, R., Trubiani, C., Baresi, L.: Autoscaling solutions for cloud applications under dynamic workloads. *IEEE Trans. on Services Computing* (2024)
17. Raith, P., Nastic, S., Dustdar, S.: Serverless edge computing - where we are and what lies ahead. *IEEE Internet Computing* **27**(3), 50–64 (2023)
18. Raith, P., Rausch, T., Dustdar, S., Rossi, F., Cardellini, V., Ranjan, R.: Mobility-aware serverless function adaptations across the edge-cloud continuum. In: *Proceedings of the International Conference on Utility and Cloud Computing*. pp. 123–132. IEEE (2022)
19. Russo, G.R., Cardellini, V., Presti, F.L.: Serverless functions in the cloud-edge continuum: Challenges and opportunities. In: *Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing*. pp. 321–328. IEEE (2023)
20. Russo, G.R., Mannucci, T., Cardellini, V., Presti, F.L.: Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In: *Proceedings of the International Conference on Pervasive Computing and Communications*. pp. 131–140. IEEE (2023)
21. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms (2017)
22. Tong, L., Li, Y., Gao, W.: A hierarchical edge cloud architecture for mobile computing. In: *Proceedings of the International Conference on Computer Communications*. pp. 1–9. IEEE (2016)
23. Vierhauser, M., Wohlrab, R., Rass, S.: Towards cost-benefit-aware adaptive monitoring for cyber-physical systems. In: *Proceedings of the Conference on Communications and Network Security*. pp. 1–6. IEEE (2022)
24. Wen, J., Chen, Z., Jin, X., Liu, X.: Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* **32**(5), 1–61 (2023)
25. Xu, D., Sun, Z.: An adaptive function placement in serverless computing. *Cluster Computing* **25**(5), 3161–3174 (oct 2022)
26. Zhang, X., Debroy, S.: Resource management in mobile edge computing: a comprehensive survey. *ACM Computing Surveys* **55**(13s), 1–37 (2023)