Joint Microservices Caching and Task Offloading Framework in VEC Based on Deep Reinforcement Learning

https://doi.org/10.3991/ijim.v17i08.37729

Ahmed S. Ghorab[1(✉)], Raed S. Rasheed[2], Hatem M. Hamad[2]
[1] University College of Applied Sciences - UCAS, Gaza, Palestine
[2] Islamic University of Gaza - IUG, Gaza, Palestine
aghorab@ucas.edu.ps

Abstract—Vehicle capacities and intelligence are rapidly increasing, which will likely support a wide range of novel and interesting applications. However, these resources are not effectively utilized. To take advantage of these invaluable capabilities in smart vehicles, they can be used in the cloud environment and can be operated through distributed computing platforms in order to benefit from their combined processing power, storage capacity, and memory resources. Vehicular edge computing (VEC) is a promising field that allows computing tasks to be transferred from cloud servers to vehicular edge servers for processing, allowing data and apps to be placed closer to vehicles (users). This paper proposes a framework that combines two modules, the first one for managing microservice caching in vehicle-mounted edge networks, such that we use cluster-based caching technique to deal with the case where similar microservices are frequently requested in VEC. The second one integrates the computational capabilities of the edge servers with the capabilities of vehicles to perform task offloading in a collaborative manner. Our solution addresses the limitations of existing edge computing platforms during peak times by combining microservices caching with computational task offloading to improve overall system performance.

Keywords—Mobile Edge Computing (MEC), task offloading, microservice caching

1 Introduction

Smart devices like smartphones, smart vehicles, and all internet of things (IoT) devices are becoming increasingly popular as the development of information technology and the growing need for improving quality of life [1]. However, IoT faces significant challenges like bandwidth limitations, storage limitations, power consumption limitations etc. To address these challenges, IBM [2] and Nokia Siemens Network [3] introduced a platform that could execute applications within a mobile base station, the term Mobile Edge Computing (MEC) was first used to characterize the execution of services at the edge of the network. It enables cloud services to be deployed close to the IoT

devices, which reduces the communication latency and the response time of the services. It can improve the quality of service for a wide range of applications such as video analytics, gaming, augmented reality, and autonomous vehicles by offloading some of the computation to the edge and thereby reducing latency and bandwidth requirements [3].

1.1 Multi-Access Edge Computing

The name "Mobile Edge Computing" has been changed to "Multi-Access Edge Computing" by the ETSI MEC industry group since 2017 to better represent the increased interest in MEC from non-cellular carriers [4]. A new technology in the 5G era makes it possible to deliver cloud and IT services in close proximity to mobile users. It makes it possible to extend cloud computing capabilities to the radio access network's edge. This new paradigm reduces backhaul use and processing at the core network while enabling the execution of delay-sensitive and context-aware applications near end users [4].

1.2 Vehicular Edge Computing (VEC)

Vehicular edge computing (VEC)[5] is a promising field that would distribute computational tasks across vehicular terminals and VEC servers to enhance vehicular services. The capacities and intelligence of vehicles are now growing quickly, which will probably support a wide variety of novel and interesting applications. Utilizing the resources of nearby vehicles allows for efficient use of network resources by reducing the load on the VEC server. The decision for partitioning and offloading tasks[5] is nontrivial, because the optimal computation offloading depends on the dynamic availability of local resources such as CPU and memory capacity within the vehicle and network bandwidth on the communication links that link the VEC servers and the vehicular terminal. These parameters change frequently due to the dynamic nature of road traffic conditions.

With the increasing number of VEC devices, a centralized deployment model is not feasible as it would consume too many resources and increase latency of communication. In a distributed infrastructure model, each device is responsible for processing some subset of tasks, and collectively these tasks form the workload. An efficient system should balance the workload among all devices such that their computational capability is used optimally to avoid unnecessary delays and overload. To achieve this, a collaborative edge computing framework that exploits the available resources in the edge nodes as well as available infrastructure in the edge servers is an essential[6]. It can combine various (heterogenous) edge computing technologies to maximize edge computing resources, such that it may combine resources from edge computing, local computing, and cloud computing to fully utilize each.

1.3 Task offloading

Tasks Offloading is one of the fundamental components of MEC. It is described as a technique that addresses the issues of computing resources, real-time, and energy consumption of mobile devices by having edge devices delegate some or all of their processing responsibilities to edge servers or cloud servers [7]. Task offloading requires coordination between the edge servers and the mobile device, which can be challenging. Several methods for effective task offloading at the edge have been proposed [8][9][10][11][12]. Traditional task offloading techniques upload the entire task to edge servers. This results in high transmission overhead and high energy consumption. Instead, there is a need for efficient techniques that offload only parts of the task to the edge server so that the edge server can process these parts and send the result back to the device. In this way, subtasks can be parallelized, which results in faster processing times and lower computational and communication overhead in the main cloud servers and data centers.

Many of the new smart devices are equipped with powerful computational capabilities and are able to support a variety of computing-intensive applications. However, these resources are not utilized [8][9]. These capabilities can be used "voluntarily" in the cloud environment and can be operated through distributed computing platforms in order to benefit from their combined processing power, storage capacity, and memory resources.

There are several challenges in building an efficient collaborative edge computing system [12]. First, it is necessary to enable heterogeneous platforms to communicate and share resources efficiently. For example, mobile devices running different operating systems need to be able to communicate with each other and access the same set of shared resources. Second, the management and operation of these distributed systems need to be coordinated in such a way that they work together seamlessly and securely to carry out the task at hand. Third, the system should be capable of supporting a wide variety of applications and workloads that are heterogeneous in nature. And finally, as the system grows in size and complexity, it should also be scalable so that new devices can be easily added while maintaining a manageable level of overhead.

1.4 Microservice caching

Microservice caching allows a local microservice to perform a task without the need to access the corresponding back-end service in the cloud [13]. This minimizes the costs incurred by end users by allowing microservice caching in a localized data center during off-peak hours. These microservices can be cached on MEC servers during off-peak hours and executed directly by the mobile device, or fetched from the cache when needed [13]. Hence, it improves the response time of the system by eliminating the need for frequent communication between the mobile device and the cloud data centers. Moreover, since the latency for communications from the mobile device to the edge server and from the edge server to the cloud is reduced, the users can enjoy a seamless experience even when the system is under heavy traffic.

However, edge servers have limited storage resources, so a wise cache allocation strategy should be used in this case, i.e., the cache needs to be selected based on the demand of each microservice [14]. Furthermore, we would like to take advantage of the massive storage capacity available in smart vehicles.

1.5 Contributions

In this research, we propose an intelligent caching framework to select the cacheable microservices based on their usage patterns to minimize memory consumption and reduce the communication overhead between the edge servers and the cloud data centers. The framework is based on clustering the vehicles based on their preferences, such that all vehicles that are interested in a specific microservice will be considered as part of one cluster and will share the same cache resources. Consequently, the edge server will just keep track of the microservices that have already been requested and the IDs of the vehicles that store them in their caches. Then, the edge server updates the global cache table. When a specific vehicle requests a microservice, it just searches for the microservice in the cache table and communicates directly with the list of vehicles that have the requested microservice. If the microservice is not found or there is no response from the hosting vehicles, it will communicate directly with the edge server to get access to the microservice from the cache of the edge cache or to pass the request to the cloud servers. Cached microservices can also be removed from the cache when they are no longer required to avoid wasting cache space.

The use of virtual clusters to implement a cache that is specific to each microservice allows for more fine-grained control over the cached data. This could be particularly useful in situations where different microservices have different cache needs and would allow for more efficient use of the cache by ensuring that only the data that is relevant to a particular microservice is stored in its cache. This could potentially improve the performance of VEC algorithms by reducing the latency associated with accessing the cached data as well as increasing the overall capacity of the cache by utilizing the combined resources of multiple servers.

Additionally, the use of clusters could also allow for the implementation of more advanced VEC techniques, such as the use of machine learning algorithms to improve the cache's performance. For example, a cluster of servers could be used to train a machine learning model that is able to predict the microservices that are likely to be accessed in the future and pre-fetch those microservices into the cache to improve performance. This could potentially improve the hit rate of the cache, as well as reduce the overall latency of the VEC algorithms.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 shows the proposed framework. Section 4 describes the deep reinforcement learning model. Finally, Section 5 is a brief conclusion to the paper.

2 Related work

Javed and Zeadally [15] provided a task-based architecture for content caching in VEC. Within this design, three key jobs are recognized. These tasks are the prediction of the content's popularity, the placement of the content in the cache, and the retrieval of the material from the cache. They provided an outline of how artificial intelligence approaches such as regression and deep Q-learning can increase the efficiency of various activities. The authors also highlighted related future research objectives in areas such as secure caching, effective sub-channel allocation for content retrieval in C-V2X, and collaborative data sharing for enhanced caching.

Sharma et al. [16]presented a macroscopic flow model for an intersection in Dublin, Ireland, using real vehicle density data. They showed that aggregated flows are extremely predictable (even though the paths of individual vehicles are not known in advance), making it possible to deploy services using vehicles' sensors. After analyzing the possibility of using car clusters as infrastructure, they presented a task-based, distributed service placement model. The distributed service scales according to the resource requirements and is robust to the changes caused by the mobility of the cluster. The goal is to minimize total processing and communication expenses. The results showed that scaling activities and finding a mobility-aware optimal placement reduces processing and communication costs.

The framework that Ko et al.[17] made for offloading computation and caching services in MEC systems takes into account how each user chooses to use services. The framework that has been proposed makes use of machine learning algorithms in order to make predictions about the service preferences of customers and choose the most effective offloading and caching approach with which to provide for those preferences. The capability of the proposed framework to successfully meet the compute offloading and service caching requirements of MEC systems is demonstrated by the results of a simulation that evaluates the performance of the framework in question.

Yu et al. [18] presented an adaptive task clustering method based on task dependence (ATCA) for task-dependent job clustering. Using data-aware scheduling, tasks with comparable characteristics are clustered and given to edge servers based on the highest value of task dependencies. It has been determined through simulation tests that the ATCA algorithm has a shorter task processing time delay and lower energy consumption than standard clustering methods. In the heterogeneous vehicle-mounted network environment, the job assigned after ATCA clustering performs well in terms of processing time delay and energy consumption, assuming all tasks are handled by the edge servers.

Thai M et al. [19] proposed a general architecture for cloud-edge computing with the goal of enabling both vertical and horizontal offloading between service nodes. They established the definitions of a service node's parent and sibling nodes, to which it may perform vertical offloading and horizontal offloading, respectively, in order to solve the issues of loop situations between service nodes. In this study, the scalability issue for large-scale cloud-edge computing systems was not solved, and the horizontal model on the level of the edge servers only was taken into consideration.

Dash S. et al. [20], presented a cloud and fog-based federated two-tier architecture, with the cloud having a higher cost and the fog having a lower cost. This design takes into account omnidirectional offloading, which allows for vertical and horizontal offloading, such as Fog to Cloud, Cloud to Fog, and Fog to Fog. Federation refers to a group of clouds working together to fulfill user requests for resources. However, they did not take into account dividing up computing work among edge nodes.

The heterogeneous mobile cloud, which combines mobile cloudlets with infrastructure-based cloudlets, was investigated by Ziyu Wu et al. [9] In such a case, mobile cloudlets can assist in offering compute offloading services when the infrastructure-based cloudlets become unavailable due to insufficient resources. For the offloaded duties, they created a centralized task scheduling method to choose trustworthy worker nodes in mobile cloudlets.

Nguyen K et al. [21], proposed a collaborative computing paradigm that efficiently offloads online heterogeneous computation tasks to parked vehicles (PVs) during peak hours. They used a container manager based on Kubernetes to integrate into the infrastructure due to its pioneering features such as auto-healing, load-balancing, and security. They did not use machine learning techniques for task offloading problems.

Hu X et al.[22], introduced a multiple parked vehicle-assisted edge computing (MPVEC) paradigm, previous works preferred to allocate workload between MEC server and single parked vehicle. A joint load balancing and offloading optimization problem is formulated to minimize the system cost under delay constraint. In this work, only the computing resources of parked vehicles (PVs) are considered to optimize the system performance.

Maftah S. et al. [23], presented a strategy for optimizing task processing and energy usage in mobile edge computing environments. The suggested system employs an intelligent offloading strategy to dynamically distribute workloads between mobile devices and adjacent edge servers based on a number of variables, including available resources, network circumstances, and energy usage.

Shaikh S. et al. [24], described 6G mobile technology's expected capabilities. The study notes that 5G technology will improve data throughput, latency, dependability, and energy efficiency. 6G technology requires sophisticated antenna technologies, new frequency bands, and new communication protocols. The authors also cover 6G applications such mobile edge computing, augmented reality, virtual reality, and haptic communication.

Raza S. et al.[25], explored the task offloading schemes by exploiting vehicle to vehicle and vehicle to infrastructure communication modes and exploiting the vehicle's under-utilized computation and communication resources, and taking the cost and time consumption into account. They presented a relay task-offloading scheme in vehicular edge computing (RVEC). According to this scheme, the tasks are offloaded in a vehicle-to-vehicle relay for computation while being transmitted to VEC servers.

To enhance the collaboration between edge and cloud resources, other researchers used joint optimization for task offloading and microservices caching in edge cloud computing environments. A three-stage heuristic approach was proposed by Chen X et al. [26] to solve the issue in polynomial time. They initially attempted to fully utilize the resources that were available in the cloud by pre-offloading as many tasks as they

could. By offloading remaining tasks and caching associated services on edge resources, their approach aimed to make full use of low-latency edge resources. Their strategy improved the performance of activities that were previously offloaded to the cloud in the final stage by re-offloading some jobs from cloud resources to edge resources. The use of heuristics has disadvantages as well. They could be quick and dirty, but they won't always result in the best choice and they might even be completely wrong. Errors in judgment and blunders might result from making rapid judgements without all the facts.

For the purpose of jointly optimizing task offloading and caching, Tang C et al. [27] studied a caching enabled task offloading in MEC. They take into account both energy consumption and response delay while solving the optimization problem. They built an alternate algorithm that depends on both the continuous variable (i.e., task offloading decision) and discrete variable (i.e., task caching decision).

Xiang L. et al.[28], investigated the collaborative task offloading issue in MEC with the aid of a dynamic caching method. To achieve a great computing resource allocation and task offloading method, they presented a two-level computing technique called joint task offloading and service caching (JTOSC). The simulation results have demonstrated that the proposed JTOSC can effectively reduce the maximum delay of all users, improve the user experience, and balance the edge load. In this work, it is assumed that all users share communication resources equally, and the inter-cell interference is ignored.

As we can note from the previous work, traditional distributed task offloading decision is mostly made by vehicles based on local states, they do not learn from each other. In addition to that, other papers used machine-learning-based methods that rely on the design of features, which leads to different feature designs that have great differences in performance.

Finally, the majority of researchers who used joint task offloading and microservices caching optimization approaches rely on heuristics methods, which frequently impose incorrect caching strategies.

3 Proposed framework

3.1 System model

To study dynamic microservice caching and task offloading, we look at the edge-cloud computing system shown in Figure 1, which is made up of different vehicles, multiple vehicular edge servers, and one cloud center.

Each vehicle has a wireless connection with the VEC server over various roadside units (RSUs).

Let $N = \{1, 2, \dots, N\}$ represents the vehicles, each one randomly requests different microservices from the VEC servers, and each one generates a task and sends an offloading request to its corresponding VEC server, which performs the offloading computing. Let $M = \{1, 2, \dots, M\}$ denotes the VEC servers. We divide time into discrete

time slots, where the operating period is slotted with an equal length t and indexed by $T = \{0,\ 1, \ldots\}$.

Vehicles has a computation task $\tau_i(t) = \{\tau_{i}_size, \tau_{i}_cpu, \tau_{i}_max\}$ to be executed at time slot t where τ_{i}_size denotes the size of the task, τ_{i}_cpu denotes the CPU cycles required to execute the task, and τ_{i}_max denotes the maximum latency allowed.

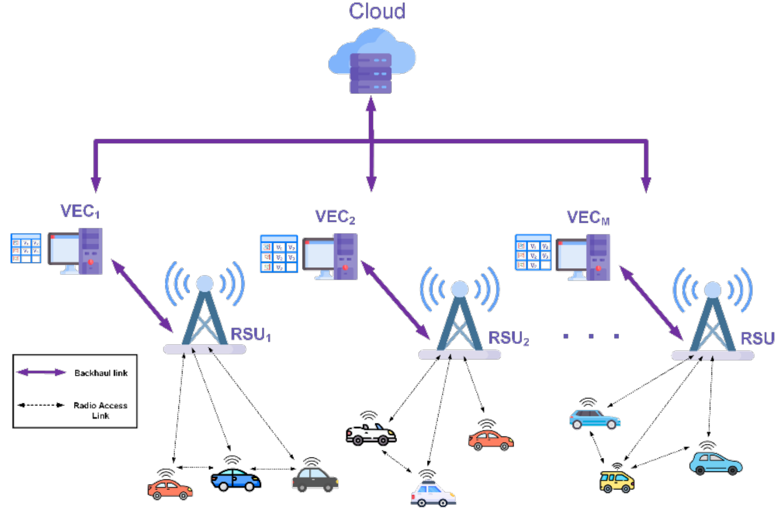

Fig. 1. General System Architecture

3.2 Task computational model

The task computation mode selection depends on the processing time requirement. Each vehicle can choose to accomplish its computation task either locally, on another vehicle, or on the VEC server.

We define the computation offloading decision of vehicle i as $c_i \in \{0,1\}$. When $c_i = 1$, it means the task is offloaded to VEC server or to other vehicle, while we set $c_i = 0$ for local computation.

If the vehicle decided to execute its task locally, the computation time can be obtained by

$$t_i^{local} = \frac{\tau_{i}_cpu}{cpu_freq} \tag{1}$$

While cpu_freq is the vehicle's CPU frequency.

The computation time of the offloaded task τ_i to VEC server can be obtained as:

$$t_i^{edge} = \frac{\tau_{i}_cpu}{p_n * edge_cpu_freq} \tag{2}$$

$p_n \in [0,1]$ is the allocated percentage of the computation resource to vehicle n in the VEC server.

3.3 Cache model

In order to accomplish the computation tasks, the VEC server can request contents (e.g., microservices) from the cloud servers through the backhaul link.

The VEC server is equipped with a cache to store popularly requested microservices so that VEC can reuse the cached microservices directly from the cache instead of obtaining the data from the cloud servers to reduce communication latency. VEC server can cache contents from cloud and use them or dispatch them to end users. If a microservice is requested, the VEC server contacts other nearby VEC servers to request the previously cached microservices.

In this paper, we assume three ways (situations) for vehicles to fetch the wanted microservices.

Local fetching: The target microservice is found in the local cache of the vehicle.

VEC fetching: The target microservice is not found at the local cache of the vehicle and is fetched from the VEC server cache or from the nearby vehicles "collaboratively."

Cloud fetching: The target microservice is neither found at the local cache of the vehicle nor at the VEC server cache. It is fetched from the cloud center.

3.4 Microservice fetch delay

Let $x_n^s(t)$ denotes the way a vehicle n fetch a microservice s at time t, such that

$$x_n^s(t) = \begin{cases} 1, & \text{if m is found at the local cache} \\ 0, & \text{if m is not found at the local cache} \end{cases}$$

Let $x_v^s(t)$ denotes the way a VEC server v fetch a microservice s at time t, such that

$$x_v^s(t) = \begin{cases} 1, & \text{if m is found at the local cache} \\ 0, & \text{if m is not found at the local cache} \end{cases}$$

So, the resultant types of delays are:

Local fetch delay: If the target microservice is cached in the local cache list, it can be accessed directly. In this way, we neglect the local fetching delay, i.e., zero. If the requested microservice is not found in local storage, the vehicle downloads the request from the VEC server.

$$\mathrm{d}_{s,n}^{\mathrm{local}}(\mathrm{t}) = 0 \tag{3}$$

VEC fetch delay: If the target microservice is cached in the local cache list, it can be accessed directly. In this way, the fetching delay of the microservice s from the VEC server at time slot t will be equal to:

$$\mathrm{d}_{s,n}^{\mathrm{VEC}}(\mathrm{t}) = \frac{\mathrm{s_{size}}}{\mathrm{R_n^{VEC}(t)}} \tag{4}$$

Where s_size is the microservice size, and $R_n^{VEC}(t)$ the transfer rate between the vehicle n and VEC server at time slot t.

Cloud fetch delay: If the target microservice is not cached on the VEC server, we need to fetch it from the cloud center. The fetching delay of the microservice s from the cloud center at time slot t will be equal to:

$$\mathrm{d}_{s,n}^{\mathrm{Cloud}}(\mathrm{t}) = \frac{\mathrm{s_{size}}}{\mathrm{R_n^{Cloud}(t)}} \tag{5}$$

Where s_size is the microservice size, and $R_n^{Cloud}(t)$ the transfer rate between the vehicle n and cloud center at time slot t.

3.5 Proposed dynamic cache management

In this paper, we propose to manage the caching of the microservices in the VEC servers and in the participating vehicles. The proposed framework places vehicles in virtual clusters according to the microservices in which they have an interest. As shown in Figure 2. This could help to ensure that the cache is being used as efficiently as possible, and could potentially improve the overall performance of the VEC system.

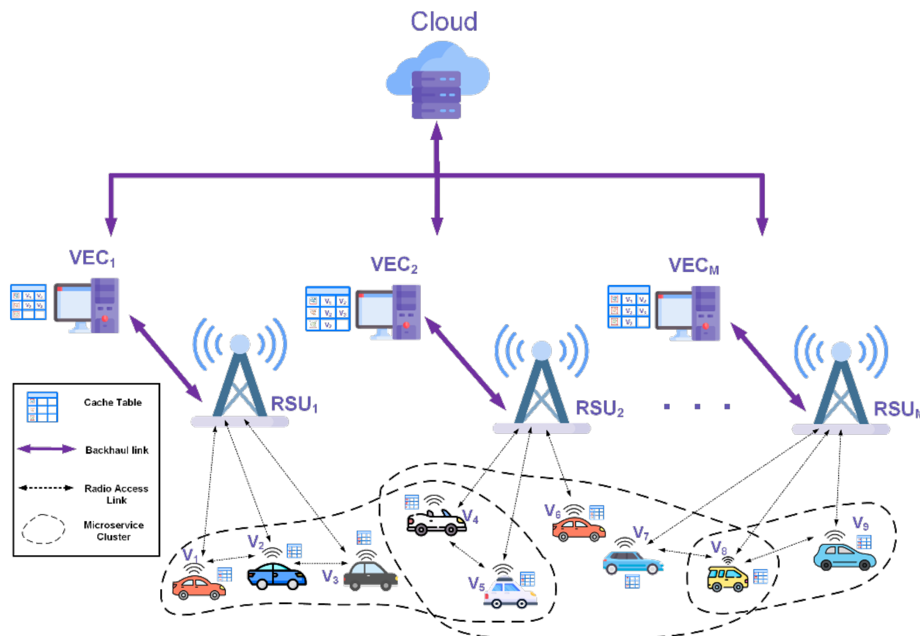

Fig. 2. The concept of virtual clusters and microservice table

Table 1 describes the abbreviations that we have used.

Table 1. List of abbreviation

Abbreviation	Definition
V	Vehicle
V_x	Current Vehicle
VEC	Vehicular Edge Computing
MS	Microservice
$LMSCT$	Local Microservice Cache Table
$GMSCT$	Global Microservice Cache Table
$VECS$	VEC Server
CS	Cloud Server
src	Source of the Microservice

The potential approach to using clusters for VEC caching involves dividing the cache into several segments, with each segment being managed by a separate cluster. This would allow for efficient distribution of caching tasks and could potentially improve the performance of the overall cache.

The idea for using clusters to improve caching techniques for microservices in a VEC system used to store and manage each individual microservice in a separate cluster (table). This would allow for each microservice to have its own dedicated cache, which could be managed and optimized separately. This could potentially improve the performance of each individual microservice, as well as allowing for more fine-grained control over the caching strategy for each microservice.

The cluster in the framework is based on the preference (i.e., the microservice itself), but if the size of the cluster increased (in different zones, for instance), the number of the active copies (which are managed by an external orchestrator) of the microservice may increase, and different copies of the microservice may be run on different locations (Edge Severs, Nodes, etc.).

Using clusters for VEC caching makes sure that all vehicles that are interested in a certain microservice are in the same virtual cluster. This makes sure that all vehicles that are interested in a certain microservice are in the same virtual cluster. For instance, vehicles $V_1, V_2, V_3, V_4, and\ V_5$ all belong to the same virtual cluster, as shown in Figure 2. Meanwhile, vehicles $V_4, V_5, V_6, V_7, and\ V_8$ all belong to different virtual cluster, and vehicles $V_8\ and\ V_9$ both belong to a third interest virtual cluster. It has also been brought to light that, given the interest in microservices, there is overlap between the three different clusters. For instance, as shown in Figure 2, $V_4\ and\ V_5$ vehicles are interested in more than one microservice. Each cluster that shares an interest in the same microservice is denoted by an individual dashed line in Figure 2. On the other hand, dashed arrows connect all vehicles linked directly within the same physical zone.

The presence of a microservice table in the vehicle implies that this microservice is preferred for this vehicle. However, if the microservice table does not exist locally and the vehicle is interested in this microservice, the vehicle contacts the VEC server to

request the microservice table, which contains all relevant microservice-related information, as in the scenario in Figure 3. This information includes a sorted list of the vehicles that are interested in the same microservice, the current location of each vehicle, and a flag indicating the existence of the microservice in the vehicle. The list of vehicles is sorted by where they are and whether or not they have the microservice in their local storage. The microservice table is stored locally in the vehicle that requested the microservice table in order to dynamically update the local microservice table whenever a new microservice is requested. The vehicle then looks up the microservice in the vehicles' list by sending a request to the nearest car that has the microservice in its storage, and so on. As in Figure 4.

Algorithm (1) describes the process of executing a microservice of interest, as shown in Figure 4. As we can see in line (5), if the MS is in the storage of the vehicle, then it will be executed directly.

The cost of loading is determined locally in the vehicle. If the cost of loading the microservice exceeds the cost of remotely executing the microservice, the microservice will be executed remotely.

If the local execution cost of the microservice is lower, the microservice will be loaded and run locally. The cost of execution will be calculated based on equations (1-5) as in Algorithm (2).

The vehicle updates the local microservice table according to the circumstances, such that it indicates whether the microservice exists locally or whether it is interested in the microservice without storing it locally. Finally, the VEC server is called in order to update the global microservice table with the information originating from the vehicle.

If the vehicle requests a microservice that the VEC server does not have, the VEC server delivers the microservice table to the vehicle to update its local microservice table. Then, the vehicle verifies the availability of the microservice with one of the vehicles in the same zone, and if the microservice is not available in any of the vehicles in the same zone, the vehicle requests the microservice from a vehicle in the same cluster by communicating with the VEC server connected directly to it, and then the VEC server connected to the vehicle that has the requested microservice via the VEC server, as depicted in Figure 4 and Figure 5.

Algorithm (1) describes the process of obtaining the MS from another vehicle. As stated in line (5), if the MS is not found within the cache of the current vehicle, it will contact the VEC server to obtain the most recent global cache table for the specific MS that it wishes to run.

The list will be sorted by the IDs of the vehicles that are nearest to the vehicle that requested the MS. Then, the vehicle will determine if the remote execution cost is higher than the local execution cost, and the vehicle request to load the microservice and execute it locally is determined.

The VEC server then changes the global microservice table and adds the vehicle request for the microservice to the microservice table.

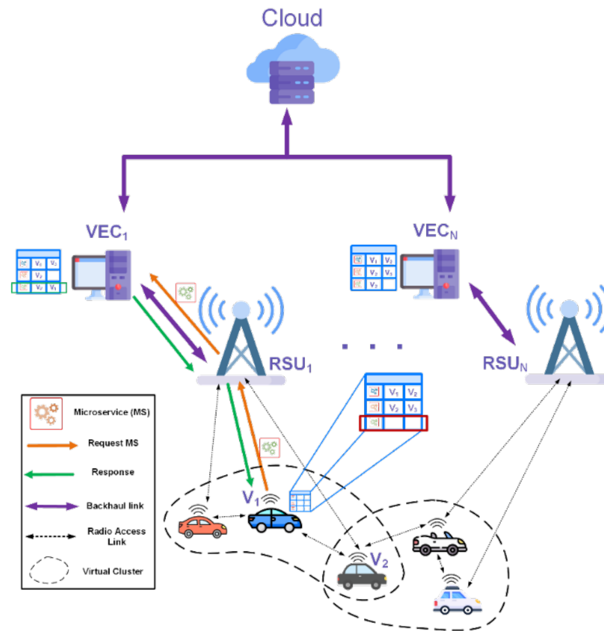

Fig. 3. If the microservice table does not exist locally as in V_1 and the vehicle is interested in this microservice, the vehicle V_1 contacts the VEC server VEC_1 to request the microservice table, which contains all relevant microservice-related information.

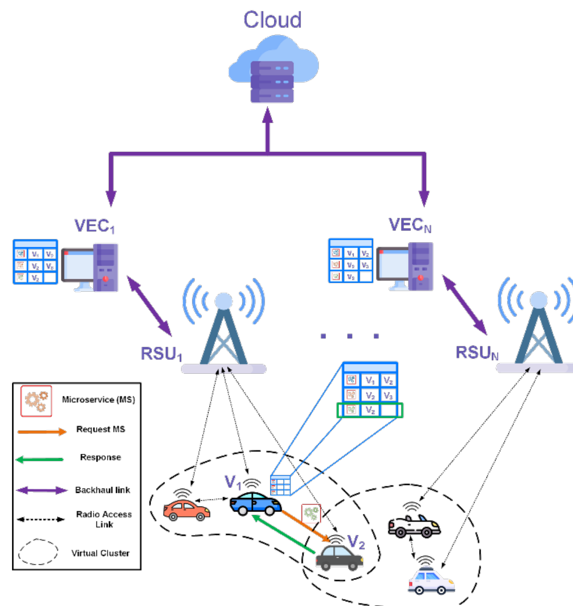

Fig. 4. If the microservice table locally as in V_1 and the vehicle is interested in this microservice, the vehicle V_1 contacts the vehicle V_2 to request the microservice.

```
Algorithm 1: Vehicle_Interest
01: Input: MS
02: Output: Finish
03: Finish ← FALSE
04: interest [MS] ← interest [MS] + 1
05: if MS in V_x Storage then
06:     call Run_MS_From(MS, V_x)
07:     Finish ← TRUE
08: else
09:     foreach Vi in cluster [MS] do
10:         if V_i in Zone [V_x] then
11:             request MS from V_i
12:             Finish ← call Run_MS(MS, VVii)
13:             if Finish then
14:               add V_x to cluster [MS]
15:               break;
16:           end if
17:         end if
18:     end foreach
19: end if
20: if NOT Finish then
21:     request MS from VECS_x
22:     Finish ← call VEC (MS, V_x)
23: end if
24: return Finish
```

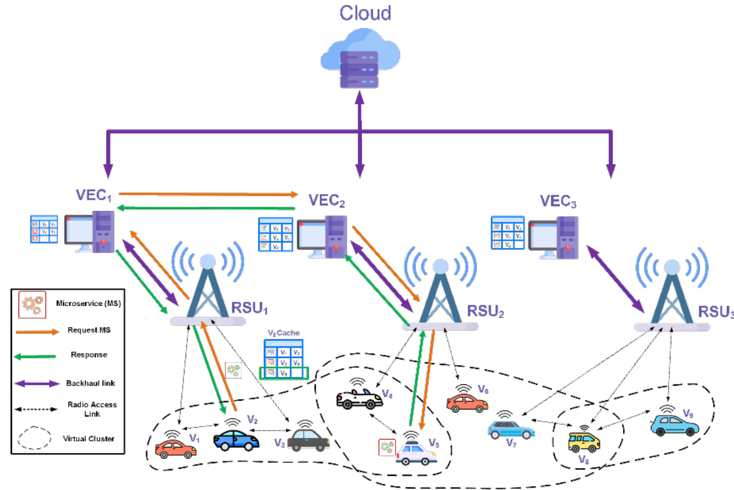

Fig. 5. If the vehicle, V_2 requests a microservice that is in V_5 in the same virtual cluster but in a different physical zone, The request will be forwarded through the connected VEC servers, and then vehicle V_5 will reply to the request through the VEC server that is connected to it.

Finally, if the microservice is not available at any of the VEC servers, and it is requested from one of the vehicles in the system, the VEC server will contact the cloud center to download the microservice to its cache, as shown in Figure 6. Algorithm (3) describes this process.

There are several potential benefits to using clusters in managing microservices, including:

Improved scalability: By using a cluster for the microservice, it is possible to scale the system up or down as needed to meet changing workloads and demand. This can help make sure that the microservices can always meet the system's requirements for performance and availability, even if traffic or work load goes up quickly.

```
Algorithm 2: Run_MS
01: Input: MS,src, V_x
02: Output: Finish
03: Finish ← FALSE
04: if MS in Source_Storage then
05:     if Cost (MS,src) < Cost (MS, V_x) then
06:         call Run_MS_From (MS,src)
07:     else
08:         call Load_MS_From (MS,src)
09:     end if
10:     update LMSCT
11:     Finish ← TRUE
12: end if
13: return Finish
```

```
Algorithm 3: VEC
01: Input: MS, V_x
02: Output: Finish
03: Finish ← FALSE
04: if MS in VECS_l Storage then
05:     Finish ← call Run_MS(MS, VECS_l)
06:     if Finish then
07:         add V_x to cluster [MS]
08:        break;
09:     end if
10: end if
11: if NOT Finish then
12:     foreach V_i in cluster [MS] do
13:        if V_i not in Zone [V_x] then
14:            request MS from V_i
15:            Finish ← call Run_MS(MS, V_i)
16:            if Finish then
17:                add V_x to cluster [MS]
18:                break;
19:            end if
20:        end if
21:    end foreach
22: end if
23: if NOT Finish then
24:     request MS from CS
25:     Finish ← call Run_MS(MS,CS)
26:     add V_x to cluster [MS]
27:     update GMSCT
28: end if
29: return Finish
```

Improved reliability: Clustering can help to improve the reliability of the microservices by providing redundancy and failover capabilities. If one VEC server in the system goes down, the other VEC servers can still provide the services that are needed. This helps make sure that the system stays up and running even if hardware or software fails.

Improved performance: Clustering the cache can also improve the performance of the microservices by allowing for the distribution of the workload across multiple servers in the system. This can help to reduce the latency associated with accessing the microservices and can also allow for faster processing of large amounts of data, particularly in real-time applications.

Improved management and maintenance: Clustering can also make it easier to manage and maintain the microservices, as the cluster can provide tools and services for managing the configuration and deployment of the microservices, as well as monitoring their performance and usage. This can help to ensure that the microservices are running

smoothly and efficiently, and it can also make it easier to identify and resolve any issues that may arise.

4 Deep reinforcement learning based decision

4.1 Reinforcement learning background

Reinforcement learning (RL) is a field of machine learning concerned with how an intelligent agent has to act in an environment to maximize some cumulative reward [29]. It is a field, which seeks to find solutions to the sequential decision in stochastic environments. At the highest level, the goal of RL is to act in an environment to optimize the reward received from the environment.

RL algorithms are important method-making for solving the Markov Decision Process (MDP) tasks [29]. In RL, agents gain rewards and change behavior policies through interacting with the surrounding environment. Therefore, the agents can obtain the highest rewards from the environment and maximize long-term accumulative rewards. Then, the goal of RL is to find the optimal policy, which is considered as the control strategy of the agent to reach the maximum expected cumulative return from each state in the state space by establishing a mapping between the state space S and action space A of the agent [29].

As shown in Figure 6, at any time t, the agent receives state S_t from state space $\mathcal{S}$ and selects an action A_t from action space $\mathcal{A}$, based on mapping policy $\pi(A_t|S_t)$. The environment generates the reward R_t and the next state S_{t+1}.

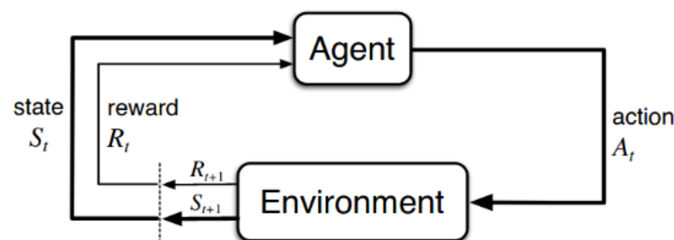

Fig. 6. RL agent and environment interaction [29]

Markov decision processes (MDP) [29]are used as a mathematical model to solve policy optimization problems. It is represented by the tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma, s_0)$, where $\mathcal{S}$ is a set of continuous states and $\mathcal{A}$ is a set of continuous actions. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the transition probability, $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function, $\gamma \in [0, 1]$ is the discount factor, and s_0 is the initial state distribution.

The goal of the RL agent is to maximize the long-term expected return from each state. In MDP, the transition probability of states is vital for estimating the value function of states and actions. In many RL tasks, however, the model of the transition probability cannot be measured precisely. As a result, model-free RL approaches [29]are commonly used to find the RL agents' best policies. Without the model of the transition

probability, the policy evaluation and improvement are calculated through the historical trajectories generated by the current policy of the agent as in Monte Carlo methods.

One of the most common model-free reinforcement learning algorithms is Q-learning [29]. It seeks to find the best action to take, given the current state. It is an off-policy RL algorithm; in this case, the learned action-value function, Q, directly approximates q^*, the optimal action-value function, independent of the policy being followed. However, convergence requires that all state-action pairs continue to be updated throughout the training process. A straightforward way to ensure this is by using an ε-greedy policy. The update rule for Q-Learning is as in "(6)".

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (6)$$

4.2 Deep reinforcement learning based decision

In many practical decision-making problems, the state $\mathcal{S}$ of the MDP are high-dimensional (e.g., images from a camera or the raw sensor stream from a robot) and cannot be solved by traditional RL algorithms[29].

In deep reinforcement learning, the agent uses a deep neural network to learn how to take actions in an environment in order to maximize a reward. The agent learns by trial and error, receiving feedback in the form of rewards or penalties for its actions. Over time, the agent learns which actions are most likely to lead to the highest rewards, and it becomes better at achieving its goals.

One of the key advantages of deep reinforcement learning is that it can learn complex behaviors that are not easily specified in traditional algorithms. This makes it well-suited for a wide range of applications, including robotics, gaming, and natural language processing.

The deployment of the microservices in our system is managed by an external orchestrator. When the microservice is deployed, the orchestrator sets the service state to "active," which may be part of the state of the environment anticipated by the RL agent that will learn the optimal policy to take actions for deploying or removing a microservice. If the microservice is active, the RL agent will automatically detect that from the current state of the microservice, and then it will use it directly. A watch mechanism inside the external orchestrator can be implemented to manage the state of the microservice and balance the load.

The deep neural network takes the state of the network as the input and produces the Q-value $Q(s(t), a(t); w)$ of each action as output, where w is the weight of the deep neural network.

The action

$$a(t) = argmax_a\, Q(s(t), a; w) \quad (7)$$

with the largest Q-value is normally chosen on the environment. Nonetheless, the action adopted may not be ideal. There is a trade-off relationship between exploration and exploitation.

Therefore, the chosen action in time epoch t is

$$a(t) = \begin{cases} argmax_a\, Q(s(t), a;\ w)\ with\ prob.\ 1-\varepsilon \\ random\ action\ with\ prob.\ \varepsilon \end{cases} \quad (8)$$

Then, the agent takes action $a(t)$ and the environment generates the reward $r(t)$ and then transits to the next network state $s(t+1)$.

The produced reward $r(t)$ and the next state $s(t+1)$, with $s(t)$ and $a(t)$, combined as a tuple $\langle s(t), a(t), r(t), s(t+1)\rangle$, are kept in a replay memory. Due to capacity constraints, the replay memory retains the most recent tuples and discards older ones. Then, a mini-batch is sampled from the replay memory to train the deep neural network so that it can increase its accuracy by learning from past events.

The aim of training the deep neural network is to minimize the loss function, which reflects the difference between the actual Q value and the estimated Q value.

$$L(w) = \mathbb{E}[-r + \gamma\, max(s', a'; w') - Q(s, a; w)] \quad (9)$$

Where $\langle s, a, c, s'\rangle$ are tuple samples from the replay buffer and w' is a neural network parameter copied from w with a certain frequency [29].

Algorithm (4) describes the steps needed to learn the agent to take the suitable action based on the current state of the system, the output of the learning process is the policy π that maps the states to the actions.

```
Algorithm 4: Pseudo code of DRL agent learning policy π
01: Initialize replay buffer capacity
02: Initialize the policy network with random weights
03: Copy the policy network, and call it the target network
04: For each episode:
05:    Initialize the starting state
06:    For each time step:
07:        Select an action based on eq (8)
08:        Execute the selected action
09:        Observe the reward and the next state
10:        Store experience in replay buffer
11:        Sample random mini batch from replay buffer
12:        Preprocess states from the mini batch
13:        Pass batch of preprocessed states to policy network
14:        Calculate loss function based on eq(9)
15:        Gradient descent updates weights to minimize loss function
16:        After specified time steps, copy weights in the policy network to the weights in the target network
```

5 Conclusions

The proposed dynamic caching framework with the elaborated deep reinforcement learning model will anticipate managing microservice caching in vehicle-mounted edge networks, such that the decisions of task offloading and microservice caching will be done by machine learning algorithms to improve the performance of the cache.

Our solution gets around the problems with existing edge computing platforms during peak times by combining microservices caching with offloading computational tasks to improve the overall performance of the system.

In our framework, we built virtual clusters of the vehicles that were interested in a microservice. The potential approach to using clusters for VEC caching involves dividing the cache into several segments, with each segment being managed by a separate cluster. This would allow for efficient distribution of caching tasks and could potentially improve the performance of the overall cache.

In order to find the optimal policy by optimizing the loss function, the system is modeled as a Markov Decision Process - MDP.

Finally, this framework could improve the hit rate of the cache and reduce the overall latency of microservices that use VEC algorithms.

6 References

[1] P. K. Deb, S. Misra, and A. Mukherjee, “Latency-Aware Horizontal Computation Offloading for Parallel Processing in Fog-Enabled IoT.”

[2] IBM Corporation, “IBM Smarter wireless networks,” 2013.

[3] Liquid Applications from Nokia Solutions and Networks and Intel® solutions for communications, “Increasing Mobile Operators’ Value Proposition With Edge Computing Turn bit pipes into smart pipes with an Intel ® architecture-based server embedded into a Nokia Solutions and Networks base station.” [Online]. Available: www.intel.com/go/commsinfrastructure

[4] S. Kekki *et al.*, “ETSI White Paper No. 28 MEC in 5G networks,” 2018. [Online]. Available: www.etsi.org

[5] S. Baidya, Y.-J. Ku, H. Zhao, J. Zhao, and S. Dey, “Vehicular and Edge Computing for Emerging Connected and Autonomous Vehicle Applications.”

[6] K. Wang, H. Yin, W. Quan, and G. Min, “Enabling Collaborative Edge Computing for Software Defined Vehicular Networks,” *IEEE Netw*, vol. 32, no. 5, pp. 112–117, Sep. 2018. https://doi.org/10.1109/MNET.2018.1700364

[7] H. Flores, P. Hui, S. Tarkoma, S. S. Y. Li, and R. Buyya, “Mobile code offloading: from concept to practice and beyond," in IEEE Communications Magazine, vol. 53, no. 3, pp. 80-88, March 2015, doi: 10.1109/MCOM.2015.7060486.,” *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80–88, 2015. https://doi.org/10.1109/MCOM.2015.7060486

[8] M. Sheikh Sofla, M. Haghi Kashani, E. Mahdipour, and R. Faghih Mirzaee, “Towards effective offloading mechanisms in fog computing,” *Multimed Tools Appl*, vol. 81, no. 2, pp. 1997–2042, Jan. 2022. https://doi.org/10.1007/s11042-021-11423-9

[9] Ziyu Wu, Lin Gui, Jiacheng Chen, Haibo Zhou, and Fen Hou, *Mobile Cloudlet Assisted Computation Offloading in Heterogeneous Mobile Cloud*. IEEE Communications Society Institute of Electrical and Electronics Engineers, 2016.

[10] S. Chen, Q. Li, M. Zhou, and A. Abusorrah, "Recent advances in collaborative scheduling of computing tasks in an edge computing paradigm," *Sensors (Switzerland)*, vol. 21, no. 3. MDPI AG, pp. 1–22, Feb. 01, 2021. https://doi.org/10.3390/s21030779

[11] F. Xu, Y. Xie, Y. Sun, Z. Qin, G. Li, and Z. Zhang, "Two-stage computing offloading algorithm in cloud-edge collaborative scenarios based on game theory," *Computers and Electrical Engineering*, vol. 97, Jan. 2022. https://doi.org/10.1016/j.compeleceng.2021.107624

[12] F. Saeik *et al.*, "Task Offloading in Edge and Cloud Computing: A Survey on Mathematical, Artificial Intelligence and Control Theory Solutions," *Task Offloading in Edge and Cloud Computing: A Survey on Mathematical, Artificial Intelligence and Control Theory Solutions. Computer Networks*, 2021. https://doi.org/10.1016/j.comnet.2021.108177

[13] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 155–158, Jul. 2018. https://doi.org/10.1109/LCA.2018.2839189

[14] J. Yao and N. Ansari, "Caching in Dynamic IoT Networks by Deep Reinforcement Learning," *IEEE Internet Things J*, vol. 8, no. 5, pp. 3268–3275, Mar. 2021. https://doi.org/10.1109/JIOT.2020.3004394

[15] M. A. Javed and S. Zeadally, "AI-Empowered Content Caching in Vehicular Edge Computing: Opportunities and Challenges," *IEEE Netw*, vol. 35, no. 3, pp. 109–115, May 2021. https://doi.org/10.1109/MNET.011.2000561

[16] K. Sharma, B. Butler, and B. Jennings, "Scaling and Placing Distributed Services on Vehicle Clusters in Urban Environments," *IEEE Trans Serv Comput*, 2022. https://doi.org/10.1109/TSC.2022.3173917

[17] S. W. Ko, S. J. Kim, H. Jung, and S. W. Choi, "Computation Offloading and Service Caching for Mobile Edge Computing Under Personalized Service Preference," *IEEE Trans Wirel Commun*, vol. 21, no. 8, pp. 6568–6583, Aug. 2022. https://doi.org/10.1109/TWC.2022.3151131

[18] Y. Yu, X. Shi, Y. Yang, X. Ren, and H. Zhao, "Clustering Algorithm Based on Task Dependence in Vehicle-Mounted Edge Networks," in *Communications in Computer and Information Science*, 2020, vol. 1267, pp. 394–400. https://doi.org/10.1007/978-981-15-9213-3_31

[19] M. T. Thai, Y. D. Lin, Y. C. Lai, and H. T. Chien, "Workload and Capacity Optimization for Cloud-Edge Computing Systems with Vertical and Horizontal Offloading," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 227–238, Mar. 2020. https://doi.org/10.1109/TNSM.2019.2937342

[20] S. Dash, R. Kumar Parida, J. Dash, A. Uddin Khan, and S. Kumar Swain, "OPTIMAL OMNIDIRECTIONAL OFFLOADING IN FEDERATED CLOUD-FOG SYSTEMS WITH COST MINIMIZATION," *J Theor Appl Inf Technol*, vol. 15, p. 17, 2022, [Online]. Available: www.jatit.org

[21] K. Nguyen, S. Drew, C. Huang, and J. Zhou, "EdgePV: Collaborative Edge Computing Framework for Task Offloading," in *IEEE International Conference on Communications*, Jun. 2021. https://doi.org/10.1109/ICC42927.2021.9500400

[22] X. Hu, X. Tang, Y. Yu, S. Qiu, and S. Chen, "Joint Load Balancing and Offloading Optimization in Multiple Parked Vehicle-Assisted Edge Computing," *Wirel Commun Mob Comput*, vol. 2021, 2021. https://doi.org/10.1155/2021/8943862

[23] S. Maftah, M. el Ghmary, H. el Bouabidi, M. Amnai, and A. Ouacha, "Optimal Task Processing and Energy Consumption Using Intelligent Offloading in Mobile Edge Computing," *International Journal of Interactive Mobile Technologies*, vol. 16, no. 20, pp. 130–142, 2022. https://doi.org/10.3991/ijim.v16i20.34373

[24] S. A. Shaikh *et al.*, "What You Should Know About Next Generation 6G Mobile Technology," *International Journal of Interactive Mobile Technologies*, vol. 16, no. 24, pp. 191–203, 2022. https://doi.org/10.3991/ijim.v16i24.35335

[25] S. Raza, M. A. Mirza, S. Ahmad, M. Asif, M. B. Rasheed, and Y. Ghadi, "A vehicle to vehicle relay-based task offloading scheme in Vehicular Communication Networks," *PeerJ Comput Sci*, vol. 7, pp. 1–20, 2021. https://doi.org/10.7717/peerj-cs.486

[26] X. Chen, T. Gao, H. Gao, B. Liu, M. Chen, and B. Wang, "A multi-stage heuristic method for service caching and task offloading to improve the cooperation between edge and cloud computing," *PeerJ Comput Sci*, vol. 8, 2022. https://doi.org/10.7717/peerj-cs.1012

[27] C. Tang, C. Zhu, X. Wei, H. Wu, Q. Li, and J. J. P. C. Rodrigues, "Task Offloading and Caching for Mobile Edge Computing," in *2021 International Wireless Communications and Mobile Computing, IWCMC 2021*, 2021, pp. 698–702. https://doi.org/10.1109/IWCMC–51323.2021.9498725

[28] X. Liu, X. Zhao, G. Liu, F. Huang, T. Huang, and Y. Wu, "Collaborative Task Offloading and Service Caching Strategy for Mobile Edge Computing," *Sensors (Basel)*, vol. 22, no. 18, Sep. 2022. https://doi.org/10.3390/s22186760

[29] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction.*

7 Authors

Ahmed S. Ghorab received his bachelor degree in computer engineering from the Islamic University of Gaza (IUG), Palestine, in 2006, and the master of science degree in computer engineering from Jordan University of Science and Technology, Jordan, in 2011. He is now a lecturer at the information technology department, University College of Applied Sciences (UCAS), Palestine. He was a teaching assistant for two years in the faculty of engineering at IUG. His current research in image processing, computer vision, data mining, mobile computing and machine learning.

Raed S. Rasheed received his bachelor degree in computer science from Applied Science University (ASU), Jordan and his master of science degree in Information Technology from the Islamic University of Gaza (IUG), Palestine. He presents several international research papers in various journals. He is a lecturer in Multimedia department, faculty of Information Technology, Islamic University of Gaza. His current professional research in blockchain, multimedia and 3D web, and interest research in computer vision and machine learning.

Hatem M. Hamad is a professor of computer engineering at the Islamic University of Gaza, specializing in web technologies, software development, and cloud computing. With over 30 years of experience in this field.

Article submitted 2023-01-15. Resubmitted 2023-03-01. Final acceptance 2023-03-02. Final version published as submitted by the authors.