Cost optimization of omnidirectional offloading in two-tier cloud–edge federated systems

Binayak Kar a,b,*, Ying-Dar Lin b, Yuan-Cheng Lai c

a Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan
b Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu, Taiwan
c Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan

A B S T R A C T

The use of the federation can exploit the advantages of cloud and edge computing technologies as the federation provides the facilities by which both can complement each other. However, certain services are needed to offload from clouds to edges, termed reverse offloading, and between edges, termed horizontal offloading. By considering the scenarios discussed above, in this paper, we propose a generic omnidirectional (OMNI) architecture of cloud–edge computing systems intending to provide vertical (edge to cloud offloading, and vice versa), and horizontal (between edges) offloading. To investigate the effectiveness of the proposed architecture in different operational scenarios, we formulate the cost optimization problem with different latency (loose, low, ultra-low) constraints. We develop an offloading algorithm using simulated annealing named the two-tier simulated annealing (TTSA) algorithm. We set two criteria for offloading: (1) offloading based on the job's size and (2) offloading based on the job's priority, irrespective of its size. The experimental results show that our proposed OMNI architecture can reduce the total cost by 7%–10%, compared to other existing architectures, and our proposed TTSA algorithm can reduce the cost by 45%–55%, compared to other existing algorithms. The average latency in OMNI architecture is relatively very less compared to other architectures.

1. Introduction

A federation is a group of service providers agreeing upon standards of operation collectively. For example, “federated cloud” facilitates the interconnection of two or more geographically separate clouds for load-balancing traffic and accommodating spikes in demand from respective users (Liaqat et al., 2017). The federation scenario is defined by various researchers differently. The collection of service providers cooperates to provide resources requested by users (Truong-Huu and Tham, 2014). For example, in cloud-A, computation is cheaper, but storage cost is high, and in cloud-B, storage cost is low but has high computation cost. In such a scenario, federation plays a key role where both the clouds agree to provide a common platform to facilitate low computation and storage costs to their customers. One service provider wholesales or rents computing resources to another service provider (Wang et al., 2017); in such a scenario by this federation, a provider acts as a user and service provider simultaneously (Goudarzi et al., 2021). For example, when the customer submits its request to cloud-A, but cloud-A does not have enough resources to serve the customer; hence it offloads (Satyanarayanan, 2015) the request to cloud-B that has the resources to serve the request. However, after the edges are re-architectures as data centers (Peterson et al., 2016), the cloud–edge federation comes into existence (Relaidonis et al., 2016), where both cloud and edge complement each other in various scenarios while providing services to their customers.

The following reasons significantly highlight the necessity of the cloud–edge and edge–edge federation. Cloud computing provides greater data storage and computing power but causes high communication latency as clouds are far away from the end users. In contrast, edge computing offers similar services with a lower communication latency as they are closer to the end user but with limited capacity and coverage (Shi et al., 2016; Shi and Dustdar, 2016). Since resources required to provide the service are limited on edge, an edge can federate with other edges and/or clouds to satisfy the users’ requests. Through this federation, the efficiency in resource utilization and enlargement of capabilities of the federated entities can increase. Fig. 1 shows a two-layer cloud–edge federated system, where the upper layer and lower layer consist of multiple clouds (like google, amazon, etc.), and edges (such as AT&T, Chunghwa telecom, etc.), respectively. The federation manager (Kanwal et al., 2014) manages the federation between two entities in these systems. In the edge–edge federation, when an edge

* Corresponding author.
E-mail addresses: bkar@mail.ntust.edu.tw (B. Kar), ydlin@cs.nctu.edu.tw (Y.-D. Lin), laiyc@cs.ntust.edu.tw (Y.-C. Lai).

https://doi.org/10.1016/j.jnca.2023.103630
Received 26 June 2022; Received in revised form 4 January 2023; Accepted 26 March 2023
Available online 1 April 2023
1084-8045/© 2023 Elsevier Ltd. All rights reserved.
does not have the required resources to handle the requests, it can offload the tasks to other edges with enough resources (Cao et al., 2020). In the edge-cloud federation, the edge can offload the workloads to the clouds that require high computation resources or extra storage space (Chekired et al., 2018). Similarly, the cloud can reverse offload the latency-sensitive tasks (Villari et al., 2016).

Depending on different types of applications, input tasks/jobs can be grouped into three categories (Hsu et al., 2015) such as ultra-low, low, and loose latency jobs. Some applications whose end-to-end latencies are a few milliseconds, even in order of 100 microseconds, are ultra-low latency jobs (Nasrallah et al., 2018), for example, industrial applications (Wollschläger et al., 2017), tactile internet (Maier et al., 2016), virtual reality (Elibamby et al., 2018), automated guided vehicles (Osseiran et al., 2015), teleurgery (Tozal et al., 2013), control traffic in smart grid (Verma et al., 2016), etc. The tasks/jobs coming with loose deadlines, i.e., in seconds, are considered loose latency jobs and are reasonable in operational cost reduction in the cloud system (Goudarzi and Pedram, 2013). The jobs whose deadline is neither tight nor too loose are considered low or medium latency jobs. To the best of our knowledge, such categorization of jobs and using all categories of jobs together in the offloading model has not been discussed before.

In this paper, we considered two approaches to categorize the above-discussed job categorization. (1) Size of the jobs (Kim et al., 2018), where small jobs are coming under ultra-low or tight latency jobs, medium size jobs are part of low latency jobs, and larger jobs are part of loose latency jobs. (2) Priority of jobs (Choudhari et al., 2018; Gao and Moh, 2018), which is set irrespective of their size, i.e., a larger job can be in the category of tight latency, and a small job can have loose latency. Job categorization based on its size is a default approach; in such a scenario, we assume a larger job requires more resources and may cause the execution to get delayed, as we reserve the resources for the small jobs and offload the large job. However, job categorization based on priority is a special approach where job size will not affect resource allocation; if the job is on priority, the required resources are available, then they must be allocated.

When highly time-sensitive jobs are given as input to an edge, and the edge cannot handle such requests, it offloads the jobs to other edges horizontally (Cao et al., 2020). Similarly, when the edge receives loose latency jobs and/or jobs that consume very high storage space, it vertically offloads to a cloud (Chekired et al., 2018). However, when a cloud receives certain applications such as microservices (Thiones, 2015), it offloads to an edge to overcome the latency and data transfer cost (Villari et al., 2016). We define the key terms used in this paper with an example as follows. Let us consider two clouds, say C₁ and C₂, and two edges, say E₁ and E₂, where clouds are in the upper tier and edges are in the lower tier.

**Definition 1.** Federation Manager (FM) is the agent that is responsible for the federation agreement between two parties to whom both the service provider will expose their information such as available resources, based on which FM will take the decision (Francescon et al., 2017). The vertical federation between a cloud and an edge is managed by the cloud–edge federation manager, whereas the edge–edge federation manager manages the horizontal federation between two edges.

**Definition 2.** Horizontal Federation: Resource sharing agreement between two nodes in the same layer. For example, E₁ and E₂.

**Definition 3.** Vertical Federation: Resource sharing agreement between two nodes in different layers. For example, E₁ in the edge layer is federated with C₁ in the cloud layer.

**Definition 4.** Horizontal Offloading: When a request for E₁ in one layer is severed by E₂ in the same layer, i.e., when E₁ in the edge layer offloaded its request to E₂ in the same layer.

**Definition 5.** Vertical Offloading: When a request is for E₁ in one layer is severed by C₁ in another layer, i.e., when E₁ offloaded its request to C₁.

**Definition 6.** Triangular Offloading: When a user of one service provider (say E₁) is served by another service provider (say C₁) and the users’ inputs are offloaded from the user to C₁ via E₁, i.e., users give input to E₁ and E₁ offloads the tasks to C₁.

**Definition 7.** Non-triangular Offloading: When a user of one service provider (say E₁) is served by another service provider (say C₁) and based on the federation agreement, the FM will offload the users’ inputs directly to C₁ without offloading via E₁.

**Definition 8.** Reverse Offloading: A vertical non-triangular offloading where a user of an upper layer node directly offloads its request to a node in the lower layer is called reverse offloading (Villari et al., 2016; Kar et al., 2022). For example, when a request for C₁ in the cloud layer is severed by E₁ in the edge layer i.e., the user of C₁ of the cloud layer offloaded its request directly to E₁ in the edge layer.

In this paper, we proposed cloud–edge federated architecture with omnidirectional offloading, where not only the edge can offload to the cloud, but also the cloud can offload back to the edge using reverse offloading, and the edges can offload to each other horizontally. By horizontal offloading, when an edge does not have enough resources to handle the requests that are time-sensitive and require less computing resources, such as ultra-low latency jobs, it can offload the tasks to nearby edges. Similarly, by vertical offloading, the edge can address the low computing resource issues by offloading the loose latency jobs to the clouds. By reverse offloading, as stated in Definition 8, the low and ultra-low latency jobs of the clouds are directly offloaded from the cloud users to the edges that are federated with the clouds, which will help to reduce both communication cost and communication latency. We set two criteria for offloading: (1) offloading based on the job's size and (2) offloading based on the job's priority, irrespective of its size. To the best of our knowledge, our work is the first to design the edge-cloud federation where offloading can be done in multiple directions (Kar et al., 2023), i.e., horizontal offloading, vertical offloading from edge-to-cloud and reverse offloading from cloud to edge. We also acknowledge the time sensitivity of different jobs and categorize them based on priority to minimize latency and communication costs. We develop an offloading algorithm using simulated annealing named the two-tier simulated annealing (TTSA) algorithm. Our detailed contributions are as follows.

1. We design a generic two-tier federated architecture enabling the clouds and edges to offload jobs omni-directionally to satisfy the users’ demand. In this architecture, not only can the edge offload to the cloud like many existing architectures, but also the cloud can reverse offload to the edge.
2. Related works

In this section, we will discuss some related work on the cloud–edge federation and how our work is different from the existing research.

Mashayekhy et al. (2014) proposed a game-theoretical model to reshape the business structure among cloud providers. These cloud providers can improve their dynamic resource scaling capabilities by establishing cooperation with the federation mechanism. In this paper, a cloud federation mechanism is used to reduce resource utilization, as a result, the profit of the providers is maximized. A capacity-sharing mechanism in a federated cloud environment is proposed in Hassan et al. (2015), whose primary focus is global energy sustainability policy. By using game theory, the paper minimizes the overall energy by capacity sharing technique and promotes long-term individual profit of the service providers. Hammoud et al. (2020) proposed a genetic algorithm to study the problem of forming highly profitable federated clouds to maintain stability among federated entities when providers join or leave federations, which might affect the Quality of Service (QoS).

Tong et al. (2016) proposed a hierarchical edge cloud architecture to improve the performance of mobile computing by leveraging cloud computing and migrating mobile workloads for remote execution in the cloud. For the efficient utilization of resources, they proposed a workload placement algorithm to decide which programs are placed on which edge cloud servers and how much computational capacity is provisioned to execute that program. A new scheduling model in the two-tier cloud-fog architecture was proposed in Chekired et al. (2018) to process the industrial Internet of things (IIoT) data. They aimed to minimize communication and data processing delays in IIoT systems by deploying multiple servers for IIoT applications at the fog layer. Ren et al. (2019) investigate the cloud edge federation, where the tasks are partially processed at the edge and the cloud server to improve their efficiency with limited communication and computation capacities. They solve this non-convex capacity optimization problem to an equivalent convex optimization problem using the Karush–Kuhn–Tucker (KKT) conditions. Samanta and Chang (2019) proposed an adaptive service offloading (ASO) scheme for the MEC platform to maximize the profit for delay-sensitive and delay-tolerant services in the presence of multiple edge devices. Ascigil et al. (2021) proposed the optimal function provisioning and resource allocation problem by considering Function-as-a-Service in the edge-cloud systems. However, since the proposed fully-centralized, optimal models require accurate prediction of upcoming requests, they use heuristic algorithms from fully centralized to fully decentralized and use single cloudlet provisioning and coordinated provisioning to make predictions accurate.

The vertical federation and horizontal federation integration are discussed in Chen et al. (2017) to determine stable cooperation partners for the federation to improve efficiency where private clouds are federated with each other horizontally. These horizontal federated clouds are federated with the public clouds vertically. Cao et al. (2020) proposed an integrated service provisioning model by considering vertical and horizontal integration between different MEC platforms. In this edge federation, they formulated a linear programming problem of the provisioning process and took a variable dimension shrinking method to solve the optimization problem.

Table 1 lists the above-discussed papers based on the federation and presents a comparison of how the contribution of our work is different from theirs. While Mashayekhy et al. (2014), Hassan et al. (2015), and Hammoud et al. (2020) are based on horizontal federation, Tong et al. (2016), Ren et al. (2019), Samanta and Chang (2019), and Ascigil et al. (2021) are based on vertical federation and Chen et al. (2017), and Cao et al. (2020) discussed both horizontal and vertical federation together. Most papers in the table discussed the cost and capacity optimization issues, and their optimization is a single-tier only, whereas the article (Tong et al., 2016) is limited to only capacity optimization and considered the two-tier optimization between cloud and edge federation with one-way vertical offloading. The articles (Chekired et al., 2018; Ren et al., 2019) have addressed the latency issue and have considered two-tier optimization but based on an edge-cloud federation scenario where the workload is offloaded from the edge layer to the cloud layer. Similarly, Samanta and Chang (2019) also addressed both latency and cost issues in a two-tier cloud–edge architecture. However, non of these articles are considered different latency sensitivity. In this paper, we classify latency into three categories depending on the type of input from the users and put certain restrictions on which type of traffic should be handled by cloud or edge. We are also considering both horizontal and vertical federations together. All the above-discussed vertical federation papers are based on one-way offloading, i.e., edge-to-cloud offloading. However, in this paper, we are considering two-way offloading where not only the edge will offload to the cloud, but also the cloud can offload to edges.

3. System model

In this section, we will discuss our proposed cloud–edge federated architecture and cost optimization problem with latency as a constraint. In Table 2, we declared the notations and variables used to describe our model and formulate the optimization problem.

**Nodes:** $C_i$ and $E_j$ represent the $i$th and $j$th nodes in tier-2 and tier-1, respectively, where the tier-1 contains all the edges and tier-2 contains all the clouds.

**Traffic:** $\lambda_i$ and $\lambda_j$ are the traffic inputs to $i$th cloud and $j$th edge, respectively. The variables $\epsilon_{ij}$ and $\eta_{ij}$ represent the vertical offloading from $j$th edge to $i$th cloud, and from $i$th cloud to $j$th edge, respectively, and horizontal offloading from $j$th edge to $j'$th edge is presented by $h_{jj'}$.

**Capacity:** $\mu_i$ and $\mu_j$ are the computing capacity of the servers in the $i$th cloud and $j$th edge, respectively. $\mu_{ij}$ is the capacity of a server used by $i$th cloud for self-computation and $\mu_{ij}$ is the computing capacity of a server in $i$th cloud used by $j$th edge. $\mu_{ij}$ is the capacity a server in $j$th edge is used by $j'$th edge, when $j = j'$, the capacity of $j$th edge server is used for self-computation. $\mu_{ij}$ is the capacity of $j$th edge server used by $i$th cloud. The notations $b_{c,v,E}$ and $b_{j,v,E}$ are communication capacity between $i$th cloud and $j$th edge, and between $j$th edge and $j'$th edge. The communication capacity between user and $i$th cloud, and user and $j$th edge are represented by $b_{u,E}$, and $b_{u,C}$, respectively.

**Cost:** The total cost of the cloud layer and edge layer are represented by $\bar{c} + \bar{d}$, respectively. The unit computing cost of clouds is $c$ and $\bar{d}$ is the same for edges. $m$, $l$, $\bar{m}$, and $\bar{l}$ are the unit communication cost from user to edge, from user to cloud, between two edges, and between edge and cloud, respectively.
The upper tier consists of clouds, and the lower tier consists of edges. The subscribers of cloud or edge can submit their requests to their respective service providers to avail of the services. In this architecture, as in Fig. 2, the edges and clouds are connected vertically, and all edges are connected horizontally. This means requests can be vertically offloaded from edges to clouds, reverse offloaded from cloud to edges, and horizontally offloaded from one edge to another. Since the clouds have unlimited capacity and coverage, we have not considered cloud-to-cloud horizontal offloading. In this paper, we consider the following assumption while modeling. Input jobs are assumed to be CPU-intensive jobs, hence do not require any input/output interruption during execution. A task cannot be partially interrupted or offloaded from one entity to another after it starts processing. Most importantly, these tasks are assumed to be independent of each other.

### Table 1

<table>
<thead>
<tr>
<th>References</th>
<th>Vertical federation</th>
<th>Horizontal federation</th>
<th>Reverse offloading</th>
<th>Cost</th>
<th>Latency</th>
<th>Latency categorization</th>
<th>Two-tier optimization</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mashayekhy et al. (2014)</td>
<td>×</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>GT</td>
</tr>
<tr>
<td>Hassan et al. (2015)</td>
<td>×</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>GT</td>
</tr>
<tr>
<td>Hammoud et al. (2020)</td>
<td>×</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>GA</td>
</tr>
<tr>
<td>Tong et al. (2016)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>SA</td>
</tr>
<tr>
<td>Chekired et al. (2018)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>SA</td>
</tr>
<tr>
<td>Ren et al. (2019)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>KKT</td>
</tr>
<tr>
<td>Samanta and Chang (2019)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>ASO</td>
</tr>
<tr>
<td>Ascigil et al. (2021)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>RAP</td>
</tr>
<tr>
<td>Chen et al. (2017)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>GT</td>
</tr>
<tr>
<td>Cao et al. (2020)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>SEE</td>
</tr>
<tr>
<td>This paper</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TTSA</td>
</tr>
</tbody>
</table>


### Table 2

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_i, 1 \leq i \leq q)</td>
<td>ith cloud in the upper tier (cloud layer) consists of q number of clouds</td>
</tr>
<tr>
<td>(E_j, 1 \leq j \leq p)</td>
<td>jth edge in the lower tier (edge layer) consists of p number of edges</td>
</tr>
<tr>
<td>(\bar{L}_{ij})</td>
<td>Traffic input to (C_i) node and (E_j) node</td>
</tr>
<tr>
<td>(u_{ij}, v_{ij}, b_{ij})</td>
<td>Vertical offloading from (E_j) node to (C_i) node, vertical offloading from (C_i) node to (E_j) node, and horizontal offloading from (E_j) node to (E_j) node</td>
</tr>
<tr>
<td>(\bar{p}_i, \bar{p}_j, \bar{\alpha}_i, \bar{\alpha}_j)</td>
<td>Computing capacity of a server in (C_i) node, capacity of a server in (C_i) used for self-computation, and capacity of a server in (C_i) used by (E_j) node</td>
</tr>
<tr>
<td>(\mu_j, \bar{\mu}_i, \bar{\alpha}_j)</td>
<td>Computing capacity of a server in (E_j) node, computing capacity of a server in (C_i) node used by (E_j) (when (j = i)) capacity of (E_j) used for self task computation, capacity of a server in (E_j) used by (C_i)</td>
</tr>
<tr>
<td>(\bar{h}<em>{E_i, E_j}, \bar{h}</em>{E_i, C_j})</td>
<td>Communication capacity between (E_i) and (E_j), communication capacity between (E_j) and (C_j)</td>
</tr>
<tr>
<td>(\bar{h}_{E_i, C_j})</td>
<td>Communication capacity between user and (E_j), communication capacity between user and (C_j)</td>
</tr>
<tr>
<td>(\overline{C}, \overline{E})</td>
<td>Unit computing cost of the cloud and edge nodes</td>
</tr>
<tr>
<td>(\overline{C}_i, \overline{E}_j)</td>
<td>Unit communication cost from user to edge, from user to cloud, between two edges, and between edge &amp; cloud</td>
</tr>
<tr>
<td>(l)</td>
<td>Total cost of the cloud layer and edge layer</td>
</tr>
<tr>
<td>(d_r)</td>
<td>Distance between a cloud and an edge node, and between two edge nodes</td>
</tr>
<tr>
<td>(d_{r}^{\overline{r}})</td>
<td>Distance between cloud and edge, and between two edges</td>
</tr>
<tr>
<td>(d^{u})</td>
<td>Distance from user to edge, and from user to cloud</td>
</tr>
</tbody>
</table>

**Latency:** \(\bar{L}_{ij}\) and \(\overline{L}_{ij}\) represent the computing latency at \(i\)th cloud, and \(j\)th edge, respectively. Whereas \(\underline{l}^{c\rightarrow E}_{ij}, \underline{l}^{E\rightarrow c}_{ij}\), and \(l^{E\rightarrow E}_{ij}\) are the communication latency from \(i\)th cloud to \(j\)th edge, from \(j\)th edge to \(i\)th cloud, and from \(j\)th edge to \(j\)th edge, respectively. In this paper, we are considering three different types of latency, i.e., **ultra-low**, **low**, and **loose** latency. \(l^{c\rightarrow E}_{ij}, l^{E\rightarrow c}_{ij}\) are the offloading latency from users to \(j\)th edge, and from users to \(i\)th cloud. The maximum latency limit for ultra-low, low, and loose latency traffics are represented by \(L_{ul}, L_{lo}, L_{ls}\), respectively.

**Distance:** \(d^{u}\) is the vertical distance between cloud and edge, and \(d^{H}\) is the horizontal distance between two edges. The distance from users to edges and clouds are \(d^{u}\), and \(d^{H}\), respectively. Speed of the light is \(c\).

### 3.1. Two-tier cloud–edge federated architecture

In this section, we will discuss our proposed generic two-tier cloud–edge federated architecture shown in Fig. 2. The upper tier consists
3.2. Traffic distribution

3.2.1. Edge layer traffic distribution

Let us assume the probability of total traffic input to the $j$th edge is 1, which includes ultra-low, low, and loose latency traffic, i.e., $\tilde{P}_j(\lambda_j)$, $P_j(\lambda_j)$, and $\tilde{P}_j(\lambda_j)$ are the probability of ultra-low, low, and loose latency traffics, respectively. The probability of offloaded ultra-low latency traffic from $j$th edge to all edges in the edge layer can be calculated as, $\tilde{P}_j(\lambda_j) = [\tilde{P}_{j1}(\lambda_j) + \tilde{P}_{j2}(\lambda_j) + \cdots + \tilde{P}_{jq}(\lambda_j)]$, where $p$ is the number of edges. Similarly, $\tilde{P}_j(\lambda_j)$ can be calculated as $\tilde{P}_j(\lambda_j) = [\tilde{P}_{j1}(\lambda_j) + \tilde{P}_{j2}(\lambda_j) + \cdots + \tilde{P}_{jq}(\lambda_j)]$ where $q$ is the number of clouds in the cloud layer. Since the low latency traffic of $j$th edge can be offloaded both horizontally to edges and vertically to clouds, then $\tilde{P}_j(\lambda_j)$ can be calculated as $\tilde{P}_j^v(\lambda_j) + \tilde{P}_j^h(\lambda_j)$, where $\tilde{P}_j^v(\lambda_j)$ is the probability of vertically offloaded traffics and $\tilde{P}_j^h(\lambda_j)$ is the horizontal offloaded traffics. $\tilde{P}_j^v(\lambda_j)$ can be calculated as $[\tilde{P}_{j1}^v(\lambda_j) + \tilde{P}_{j2}^v(\lambda_j) + \cdots + \tilde{P}_{jq}^v(\lambda_j)]$, and $\tilde{P}_j^h(\lambda_j)$ as $[\tilde{P}_{j1}^h(\lambda_j) + \tilde{P}_{j2}^h(\lambda_j) + \cdots + \tilde{P}_{jq}^h(\lambda_j)]$. Then the vertically offloaded total traffic from $j$th edge to $j$th cloud will be, $v_{ij} = \tilde{P}_{j1}^v(\lambda_j) + \tilde{P}_{j2}^v(\lambda_j) + \cdots + \tilde{P}_{jq}^v(\lambda_j)$. Total horizontally offloaded traffic from $j$th edge to $j$th edge will be, $h_{ij} = \tilde{P}_{j1}^h(\lambda_j) + \tilde{P}_{j2}^h(\lambda_j) + \cdots + \tilde{P}_{jq}^h(\lambda_j)$. In $h_{ij}$, if $j = j’$ then there will be no horizontal offloading.

3.2.2. Cloud layer traffic distribution

Let us assume the probability of total traffic input to the $i$th cloud is one that includes ultra-low, low, and loose latency traffics, i.e., $P_i(\lambda_i)$, $\tilde{P}_i(\lambda_i)$, and $\tilde{P}_i(\lambda_i)$ are the probability of ultra-low, low latency, and loose latency traffic input to $i$th cloud, respectively. According to our architecture, the ultra-low latency traffics are processed at the edge layer due to their time sensitiveness; hence cloud always offloads such requests to the edges. $P_{ij}(\lambda_i)$ represents probability of offloaded ultra-low latency traffic from $i$th cloud to $j$th edge. Then the total probability of offloaded ultra-low latency traffic from $i$th cloud $C_i$ to the edge layer will be, $P_i(\lambda_i) = P_{i1}(\lambda_i) + P_{i2}(\lambda_i) + \cdots + P_{iq}(\lambda_i)$. $P_{ij}(\lambda_i)$ represents probability of offloaded low latency traffic from $i$th cloud to $j$th edge and $\tilde{P}_i(\lambda_i)$ is the probability of same traffic type for $i$th cloud for self-computation. Then the probability of total low latency traffic input to a $i$th cloud will be, $\tilde{P}_i(\lambda_i) = \tilde{P}_{i1}(\lambda_i) + \tilde{P}_{i2}(\lambda_i) + \cdots + \tilde{P}_{iq}(\lambda_i)$. Then the total traffic offloaded from $i$th cloud to $j$th edge can be calculated as, $\tilde{v}_{ij} = \tilde{P}_{i1}(\lambda_i) + \tilde{P}_{i2}(\lambda_i) + \cdots + \tilde{P}_{iq}(\lambda_i)$.

3.3. Latency calculation

In our model, we assume both the clouds and edges consist of multiple servers, and each server has an equal capacity. However, the number of servers in a cloud is relatively high compared to an edge. Hence, a cloud has more computational capacity than an edge. To calculate the computation latency of the nodes, we apply the M/M/$k$ queuing model, where $k$ is the number of servers in the node. The value of $k$ in the edges is kept fixed as the capacity of the edge is limited. However, since the capacity of the cloud is unlimited, we have considered the value of $k$ is dynamic. The communication latency is calculated by the M/M/1 queuing model.

3.3.1. Computational latency calculation

Based on M/M/$k$ model, the computational latency of $i$th cloud and $j$th edge are calculated in Eqs. (1) and (2), respectively, as follows.

$$l_i^C = \frac{C(k_i^C) \cdot J_i^C \cdot \mu_i}{k_i^C \cdot \rho_i^C - J_i^C} + \frac{1}{\mu_i}, \text{ where } J_i^C < k_i^C \cdot \mu_i,$$

$$l_j^E = \frac{C(k_j^E) \cdot J_j^E \cdot \mu_j}{k_j^E \cdot \rho_j^E - J_j^E} + \frac{1}{\mu_j}, \text{ where } J_j^E < k_j^E \cdot \mu_j,$$

where $k_i^C$ and $k_j^E$ are the number of servers present in an edge, and cloud node, respectively. $J_i^C = \lambda_i - \sum_{j=1}^{n} \tilde{v}_{ij} + \sum_{j=1}^{n} v_{ij}$ and $J_j^C = \lambda_j - \sum_{i=1}^{n} \tilde{v}_{ij} + \sum_{i=1}^{n} v_{ij}$. $C(k, \lambda, \mu)$ is the Erlang C formula (Shorttle et al., 2018), where $C(k, \lambda, \mu) = \frac{1}{\sum_{k=0}^{k-1} \frac{k!}{\mu^k} + \frac{\lambda}{\mu} \sum_{k=1}^{\infty} \frac{1}{(k-1)!} \frac{1}{\mu^{k-1}}}$ and $\rho = \lambda/\mu$.

3.3.2. Communication latency calculation

From the users to the cloud/edge nodes, initial communication latency can be calculated as follows. The communication latency from the user to $j$th edge and from the user to $i$th cloud is calculated in Eqs. (3), and (4), respectively. Similarly, the communication latency from $i$th cloud to $j$th edge, from $j$th edge to $i$th cloud, and from $j$th edge to $j’$th edge are calculated in Eqs. (5), (6), and (7), respectively, as follows.

$$l_{ij}^E = \frac{1}{b_{ij}} - \frac{1}{\mu_i}, \text{ where } \lambda_i < b_{ij} \cdot C_i,$$

$$l_{ij}^E = \frac{\mu_i}{\rho_i} \cdot \lambda_i, \text{ where } \lambda_i < b_{ij} \cdot C_i,$$

$$l_{ij}^{E-C} = \frac{1}{b_{ij}^C} - \lambda_i, \text{ where } \lambda_i < b_{ij}^{C_i} \cdot C_i^{E-C},$$

$$l_{ij}^{E-C} = \frac{1}{b_{ij}^C} - \lambda_i, \text{ where } \lambda_i < b_{ij}^{C_i} \cdot C_i^{E-C},$$

$$l_{ij}^{E-C} = \frac{1}{b_{ij}^C} - \lambda_i, \text{ where } \lambda_i < b_{ij}^{C_i} \cdot C_i^{E-C},$$

$$l_{ij}^{E-C} = \frac{1}{b_{ij}^C} - \lambda_i, \text{ where } \lambda_i < b_{ij}^{C_i} \cdot C_i^{E-C},$$

3.4. Triangular offloading problem

Triangular offloading is the case where users give their input to a node (service provider), say $S_0$. Node $S_0$ determines whether the input is appropriate to handle or not based on its available capacity and other constraints. If yes, then the input is handled by the node itself. Otherwise, $S_0$ will determine another service node say $S_n$, which has a federation agreement with $S_0$ and can handle the input. Then $S_n$ will offload the requests to $S_1$. This communication from the user to $S_0$ and then from $S_0$ to $S_1$ (i.e., user $\rightarrow S_0 \rightarrow S_1$) is called triangular offloading. In this section, for such triangular offloading, we present the optimization problem.
3.4.1. Objective of the edge layer

Let $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, and $x_6$ be the variables in Eq. (8), presenting different costs of nodes in the edge layer.

\[
\begin{align*}
& x_1 = \sum_{p=1}^{P} \hat{a}_j \cdot m_j, \\
& x_2 = \sum_{p=1}^{P} \sum_{j=1}^{J} k_j \cdot \hat{b}_{j, p} \cdot \hat{c}_p, \\
& x_3 = \sum_{p=1}^{P} \sum_{j=1}^{J} \hat{h}_{j, p} \cdot \hat{m}, \\
& x_4 = \sum_{p=1}^{P} \sum_{j=1}^{J} \hat{k}_j \cdot \hat{p}_{j, p} \cdot \hat{c}_p, \\
& x_5 = \sum_{p=1}^{P} \sum_{j=1}^{J} \hat{l}_{j, p} \cdot \hat{h}, \\
& x_6 = \sum_{p=1}^{P} \sum_{j=1}^{J} \hat{c}_j \cdot \hat{p}_{j, p} \cdot \hat{c}_p.
\end{align*}
\]  

(8)

Where $x_1$ shows the communication cost from the users to edges. The total self-computing cost of the edges is presented by $x_2$. The total communication cost between edges is shown by $x_3$. The $x_4$ shows the total computing cost of the edges while computation is done by other edges. The total communication cost from the edges to the clouds is presented by $x_5$. The $x_6$ presents the total computing cost of the edges while computation is done by the clouds. Then the objective function of the edges is to minimize $\tilde{\ell}$, where

\[
\tilde{\ell} = x_1 + x_2 + x_3 + x_4 + x_5 + x_6,
\]

subject to,

\[
\begin{align*}
& \mu_j^P + \tilde{\ell}_j < L_{al}, \quad (a) \\
& \mu_j^P + \tilde{\ell}_j < L_{il}, \quad (b)
\end{align*}
\]  

(10)

\[
\begin{align*}
& \mu_j^P + \tilde{\ell}_j < L_{al}, \quad (a) \\
& \mu_j^P + \tilde{\ell}_j < L_{il}, \quad (b)
\end{align*}
\]  

(11)

\[
\begin{align*}
& \mu_j^P + \tilde{\ell}_j < L_{li}, \quad (c)
\end{align*}
\]  

(12)

\[
0 < \hat{p}_{j, p} \leq \mu_j, \forall j = 1, \ldots, p.
\]

(13)

The Eqs. (10)–(12) present the latency constraints for the inputs given to all edges from users, where Equations in (10) are for ultra-low latency, Equations in (11) are for low latency, and Eq. (12) is for loose latency traffics. The inequalities in Eqs. (10)(a) and (11)(a) show the communication latency from the user to $j$th edge plus the computing latency at $j$th edge must be less than the ultra-low latency limit, and low latency limit, respectively. Eqs. (10)(b) and (11)(b) are the cases where the user submitted the jobs to $j$th edge, and $j$th edge offloaded the job to $j$th edge. The inequalities in Eqs. (10)(b), and (11)(b) show the sum of the user to $j$th edge communication latency, $j$th edge to $j$th edge communication latency and the computing latency at $j$th edge must be less than the ultra-low latency limit, and low latency limit, respectively. Eqs. (11)(c), and (12) are the cases where the user submitted the jobs to $j$th edge, and $j$th edge offload the job to $j$th cloud. The inequalities in Eqs. (11)(c) and (12) show the sum of the user to $j$th edge communication latency, $j$th edge to $j$th cloud communication latency and the computing latency at $j$th cloud must be less than the low latency limit, and loose latency limit, respectively. The constraint in Eq. (13) presents an edge that can neither compute all its received requests by itself nor offload them entirely to others. The sum of self-computation and horizontal offloading capacity plus vertical offloading capacity must be less than the total input to the edges presented in Eq. (14).

3.4.2. Objective of the cloud layer

Let $y_1$, $y_2$, $y_3$, and $y_4$ be the variables in Eq. (15), presenting different costs of nodes in the cloud layer.

\[
\begin{align*}
& y_1 = \sum_{q=1}^{Q} \hat{a}_q \cdot \hat{m}, \\
& y_2 = \sum_{q=1}^{Q} \sum_{k=1}^{K} \hat{b}_k \cdot \hat{c}_q \cdot \hat{e}_k, \\
& y_3 = \sum_{q=1}^{Q} \sum_{k=1}^{K} \hat{h}_k \cdot \hat{m}, \\
& y_4 = \sum_{q=1}^{Q} \sum_{k=1}^{K} \hat{k}_k \cdot \hat{p}_{j, k} \cdot \hat{c}_q \cdot \hat{e}_k.
\end{align*}
\]  

(15)

Where $y_1$ shows the communication cost from the users to the clouds. The total self-computing cost of the clouds is presented by $y_2$. The total communication cost from the clouds to the edges is presented by $y_3$. The $y_4$ presents the total computing cost of the clouds while computation is done by the edges. Then the objective function of the clouds is to minimize $\tilde{\ell}$, where

\[
\tilde{\ell} = y_1 + y_2 + y_3 + y_4,
\]

subject to,

\[
\begin{align*}
& \mu_i^C + \tilde{\ell}_i < L_{al}, \quad (a) \\
& \mu_i^C + \tilde{\ell}_i < L_{il}, \quad (b)
\end{align*}
\]  

(16)

\[
\begin{align*}
& \mu_i^C + \tilde{\ell}_i < L_{al}, \quad (a) \\
& \mu_i^C + \tilde{\ell}_i < L_{il}, \quad (b)
\end{align*}
\]  

(17)

\[
\begin{align*}
& 0 < \hat{p}_{j, p} \leq \mu_i, \forall i = 1, \ldots, p.
\end{align*}
\]  

(18)

The Eqs. (17)–(19) presents the latency constraints for the inputs given to all clouds from the users, where Eq. (17) is for ultra-low latency, equations in (18) are for low latency, and Eq. (19) is for loose latency traffics. Eqs. (17), and (18)(b) are the cases where the user submitted the jobs to $i$th cloud, and $i$th cloud offloaded the jobs to $j$th edge. The inequalities in Eqs. (17) and (18)(b) show the sum of the user to $i$th cloud communication latency, $i$th cloud to $j$th edge communication latency and the computing latency at $j$th edge must be less than the ultra-low latency limit, and low latency limit, respectively. The inequalities in Eqs. (18)(a) and (19) show the user to $i$th cloud communication latency plus the computing latency at $i$th cloud must be less than the low latency limit, and loose latency limit, respectively. The constraints in Eq. (20) present any cloud can neither compute all its request by itself nor offload them entirely to others, provided all the traffics to the cloud are not loose latency traffics. The sum of self-computation and vertical offloading capacity must be less than the total input to the clouds discussed in Eq. (21).

3.5. Non-triangular offloading problem

As discussed in Section 3.4, in triangular offloading, when two service nodes $S_u$ and $S_l$ have a federation agreement, $S_u$ receives the request from the user and offload to $S_l$. However, in non-triangular offloading, if $S_u$ and $S_l$ have a federation agreement, then the user of $S_u$ can directly submit its request to $S_l$ by the FM. This type of offloading is called non-triangular offloading. In this subsection, we modify our previously proposed optimization problem for such non-triangular offloading.
3.5.1. Non-triangular latency calculation

In a non-triangular offloading scenario, the communication latency is estimated as follows. The vertical offloading latency from the user to the cloud tier denoted as \( t_{\text{old}} \), where \( \theta = \frac{1}{2} \sum_{j=1}^{p} h_{j} \cdot m_{j} \). The horizontal offloading latency from the user to the edge tier denoted as \( t_{\text{edge}} \), where \( \phi = \frac{1}{2} \sum_{j=1}^{p} v_{j} \cdot m_{j} \). Therefore, the communication latency is estimated as \( \frac{1}{2} \sum_{j=1}^{p} h_{j} \cdot m_{j} + \frac{1}{2} \sum_{j=1}^{p} v_{j} \cdot m_{j} \), where \( h_{j} \leq h_{\text{CE}} \). The reverse offloading latency from the user to the edge tier, where \( \phi = \frac{1}{2} \sum_{j=1}^{p} v_{j} \cdot m_{j} \). Therefore, the communication latency is estimated as \( \frac{1}{2} \sum_{j=1}^{p} h_{j} \cdot m_{j} + \frac{1}{2} \sum_{j=1}^{p} v_{j} \cdot m_{j} \), where \( h_{j} \leq h_{\text{CE}} \).

3.5.2. Modified objective of the edge layer

The new cost in the edge layer required the following cost variables, including some variables from Eq. (8).

\[
\begin{align*}
\hat{x}_{i} & = \sum_{j=1}^{p} \sum_{i,j \neq j} h_{i,j} \cdot m_{i,j} + \frac{\theta}{\theta + \phi} \\
\hat{y}_{i} & = \sum_{j=1}^{p} \sum_{i,j \neq j} v_{i,j} \cdot m_{i,j} + \frac{\phi}{\theta + \phi} \\
\end{align*}
\]

subject to,

\[
\begin{align*}
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\end{align*}
\]

The Eqs. (24) and (25)(a) are the cases where jobs from the user of \( j \)th edge are directly offloaded to \( j \)th edge. The inequalities in Eqs. (24) and (25)(a) present the sum of the user to \( j \)th edge communication latency and, the computing latency at \( j \)th edge must be less than the ultra-low latency limit, and the low latency limit, respectively. Eqs. (25)(b) and (26) are the cases where jobs from the user of \( j \)th edge are directly offloaded to \( j \)th cloud. The inequalities in Eqs. (25)(b) and (26) present the sum of the user to \( j \)th cloud communication latency, and computing latency at \( j \)th cloud must be less than the low latency limit, and loose latency limit, respectively.

3.5.3. Modified objective of the cloud layer

The new cost in the cloud layer required the following cost variables, including some variables from Eq. (15).

\[
\begin{align*}
\gamma_{x} & = \sum_{j=1}^{p} \sum_{i,j \neq j} h_{i,j} \cdot m_{i,j} + \frac{\theta}{\theta + \phi} \\
\gamma_{y} & = \sum_{j=1}^{p} \sum_{i,j \neq j} v_{i,j} \cdot m_{i,j} + \frac{\phi}{\theta + \phi} \\
\end{align*}
\]

subject to,

\[
\begin{align*}
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\end{align*}
\]

After analyzing both the triangular offloading problem in Section 3.4 and the non-triangular offloading problem in Section 3.5, we found both the offloading scenarios have their advantages as well as disadvantages. For example, triangular offloading increases the communication latency, whereas non-triangular offloading reduces the communication latency but increases the burden on the federation manager to make an optimal decision. Hence, we adopted both offloading scenarios in our OMNI architecture. For the edge layer, we consider the triangular offloading as in Eq. (9), where the edge will offload the task to other edges horizontally and the cloud vertically. Because in this triangular offloading, the communication latency will not make more difference than the non-triangular offloading due to the short distance from the user to the edges and between edges. However, we consider the non-triangular offloading for the cloud layer as in Eq. (28), where the cloud will reverse offload the time-sensitive tasks to the edges. By this reverse offloading, the communication latency of non-triangular offloading will be relatively less as compared to triangular offloading since the distance "from user to cloud and cloud to edge (triangular)" is much higher than simply "from user to edge (non-triangular)." Since the main goal of our model is to reduce the total cost of the system, with given latency as the constraint, the modified problem is as follows. The modified objective function is to minimize the total system cost, i.e., minimize \( \hat{t}_{\text{total}} \), where

\[
\hat{t}_{\text{total}} = a \cdot \hat{t} + (1 - a) \cdot \hat{t}_{1},
\]

subject to,

\[
\begin{align*}
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\mu_{\text{CE}} + \hat{t}_{i} & < L_{\text{CE}}, \\
\end{align*}
\]

4. Problem formulation

In the solution, we describe the simulated annealing (SA) algorithm (Johnson et al., 1989), which performs a probabilistic technique to find a globally optimal solution. Specifically, it is a meta-heuristic approach to approximate global optimization in an ample search space for an optimization problem and is used when the search space is discrete.

4.1. Two-tier simulated annealing

The two-tier simulated annealing (TSSA) process is a modified version of SA described in Algorithm 1. In each iteration, we generate a new state \( x_{\text{new}} \) from the previous state \( x_{\text{old}} \) and compute the cost for both the tiers using \( \text{Cost}_{t} \), i.e., in Eq. (9) and \( \text{Cost}_{t} \), i.e., in Eq. (28). The differences between the old and new costs of the edge tier and cloud tier denoted as \( \Delta_{1} \) and \( \Delta_{2} \), respectively, are computed. The \( x_{\text{new}} \) will be immediately accepted if \( \Delta_{1} \leq 0 \) and \( \Delta_{2} \leq 0 \). If \( \Delta_{1} \leq 0 \) and \( \Delta_{2} > 0 \), then we have a probability \( e^{-\Delta_{1}/T} \) to accept \( x_{\text{new}} \), where \( T \) is the simulated temperature and decreases in each step of the iteration by a cooling parameter \( a \) (0 < \( a < 1 \)). However, if \( \Delta_{1} > 0 \) and \( \Delta_{2} \leq 0 \), we
Algorithm 1 TTSA Algorithm

Initially: Randomly generate initial solution \( x_{\text{old}} \)

Set: \( T \leftarrow T_{\text{max}} \)

while \( T > T_{\text{min}} \) do

Generate new solution \( x_{\text{new}} \)

\[ \Delta_1 = \text{Cost}_1(x_{\text{new}}) - \text{Cost}_1(x_{\text{old}}) \]

\[ \Delta_2 = \text{Cost}_2(x_{\text{new}}) - \text{Cost}_2(x_{\text{old}}) \]

if \((\Delta_1 < 0) \& (\Delta_2 < 0)\) then

accept \( x_{\text{old}} \leftarrow x_{\text{new}} \)

else

if \((\Delta_1 < 0) \& (\Delta_2 > 0)\) then

accept \( x_{\text{old}} \leftarrow x_{\text{new}} \) with probability \( e^{-\Delta_2/T} \)

else

if \((\Delta_1 > 0) \& (\Delta_2 < 0)\) then

accept \( x_{\text{old}} \leftarrow x_{\text{new}} \) with probability \( e^{-\Delta_1/T} \)

else

if \((\Delta_1 > 0) \& (\Delta_2 > 0)\) then

accept \( x_{\text{old}} \leftarrow x_{\text{new}} \) with probability \( e^{-(\Delta_1+\Delta_2)/T} \)

end

end

end

\( T \leftarrow a \cdot T \)

end

can accept \( x_{\text{new}} \) with probability \( e^{-\Delta_1/T} \). The \( x_{\text{new}} \) will acceptable with probability \( e^{-(\Delta_1+\Delta_2)/T} \) if \( \Delta_1 > 0 \) and \( \Delta_2 > 0 \). The length of the iterations in SA is determined by the initial temperature \( T_{\text{max}} \), the terminating temperature \( T_{\text{min}} \), and the cooling parameter \( a \). At the beginning, \( T \) equals to \( T_{\text{max}} \). \( T \) then decreases in each iteration, when \( T = T_{\text{min}} \) the SA iterations terminate.

This algorithm applies randomness with some restrictions for the initial solution, which is initialized as follows. As discussed in our proposed architecture, our input jobs are three different types. Only edges handle the ultra-low latency jobs, only clouds handle loose latency jobs, and both edges and clouds can handle low latency jobs. For a cloud, when the inputs are loose latency jobs, they will have kept by the cloud, whereas if the job is ultra-low latency, it will offload to any edge randomly. For low latency jobs, the cloud will have kept 70 percent by itself and offload the rest to the edges randomly. For an edge, if the inputs are loose latency jobs, it will offload randomly to the clouds. If the inputs are low latency jobs, then 50 percent of jobs are kept on the edge, and the other 50 percent will be offloaded to other edges and clouds. If the inputs are ultra-low latency jobs, then 70 percent will be kept by the edge, and the other 30 percent will be randomly offloaded to other edges.

We use three processes (swapping, reversion, and insertion) in this simulated annealing to make the offloading decision. In the swapping process, two federated nodes can swap their jobs with each other. And in the insertion process, a job from one node is taken and inserted in another node. While invoking these processes, we impose some restrictions not to violate offloading criteria set in the systems.

1. In the swapping process, “two loose latency jobs/two low latency jobs/one low and one loose latency jobs” between two clouds is applicable if and only if those two jobs are offloaded from edges. Non-offloaded jobs cannot be swapped between two clouds as there is no cloud-to-cloud offloading. The swapping of two low latency jobs between an edge and a cloud is applicable, provided the selected job in the edge is not offloaded from any other clouds. The swapping can be performed between two edges if the following condition holds. (a) if both the jobs are low latency jobs, (b) if both the jobs are ultra-low latency jobs, and (c) if one is a low latency job and the other is an ultra-low latency job.

2. In the reversion process, after two jobs are selected for swapping, the necessary swapping condition (as discussed above) must be satisfied between intermediate jobs in the intermediate nodes. If there is any violation of the swapping rule between two intermediate nodes, then swapping of those jobs will not be performed.

3. In the insertion process, a loose latency job cannot be inserted in an edge, and an ultra-low latency job cannot be inserted in a cloud. A loose/low latency job, initially assigned to one cloud, can be inserted into a new cloud if and only if the job was offloaded from an edge.

Consequently, like the SA process, the TTSA also has no bounded time complexity. Instead, its time complexity is determined by the cooling parameter \( a \).

5. Numerical results

5.1. Experiment setup

For the experiment, we considered a cloud–edge federated network consisting of five clouds, with five edges geographically distributed. The capacity of a server on edge is 2 GB, and each edge consists of 10 servers. The capacity of a server in the cloud is 4 GB, and the number of servers in the cloud node is unlimited. The maximum distance from the user to an edge, the user to a cloud, between two edges, and between an edge and a cloud are 10 KM, 1000 KM, 100 KM, and 1000 KMs, respectively. The size of the input jobs is taken randomly between 1 KB to 250 MB. Based on the size, the jobs are categorized into three groups. The unit cost of communication and computation is considered in terms of money. The detailed parameters for our experiment are discussed in Table 3. We conducted simulations to evaluate the performance of OMNI architecture and the proposed TTSA algorithm. To calculate the total cost in Eq. (31), we set the value of \( a \) to 0.5 since the job inputs given to both the edge and cloud layers are equal, and the cooling parameter \( a \) value is set to 0.99. Our simulation runs on a Windows 11 operating system running on an i7 processor with 16 GB RAM. We carried out our simulation by using Python and compared it with CPLEX (Cplex, 2009) for the optimal result. Fig. 3 shows the cost analysis of TTSA and CPLEX results for omnidirectional architecture. This result is of cost analysis based on job sizes, and the input jobs were taken randomly from 490 to 510 KB. Fig. 3 shows the total cost result from the TTSA algorithm and CPLEX result are nearly similar.

5.2. Performance analysis

In this section, we compared the performance with three other architectures: (1) zero offloading model (ZOM), (2) edge-to-cloud vertical
offloading (E2CVO) as in Tong et al. (2016), and (3) edge-to-edge horizontal offloading (E2EHO) as in Cao et al. (2020). In this experiment, we considered two types of input scenarios. First, the offloading is based on the job’s size, where a small size job is treated as an ultra-low latency job that an edge must process, and a large job is treated as a loose latency job that a cloud must process. (2) Second, offloading is based on the job’s priority, irrespective of its size. A priority value (low, mid, high) was randomly assigned to each job, irrespective of their size. In this scenario, a job of any size can be an ultra-low, low, or loose latency job.

5.2.1. Total cost analysis – OMNI vs. Others

Figs. 4 and 5 show the cost analysis of different architectures with the increase in the number of input jobs. Fig. 4 shows the system performance where the offloading decision of the jobs is considered based on their priority irrespective of their size. In contrast, Fig. 5 shows the offloading decision of the jobs is taken based on the size of the jobs as given in Table 3. In both figures, we presented the total cost of the systems in Figs. 4(a) and 5(a), and the intermediate results are the edge layer cost in Figs. 4(b) and 5(b), and the cloud layer cost in Figs. 4(c) and 5(c). If we consider the total cost as given presented in Figs. 4(a) and 5(a), in both cases, our OMNI architecture saves the total cost by 7%–10%, 20%–30%, and 25%–40% compared to the E2CVO, E2EHO, and ZOM architectures, respectively, because of its horizontal, vertical, and reverse offloading mechanisms. Due to reverse offloading, the highly time-sensitive jobs of clouds are redirected to edges, which reduces the communication cost. Due to horizontal federation, the edge extends its computing capacity in the edge layer. The E2CVO saves more cost than ZOM and E2EHO as the edges offload the loose latency and some low latency jobs to the clouds, reducing the storage burden on the edges and making the computation faster as the cloud has unlimited computing resources. Let us compare the ZOM and E2EHO performance. The E2EHO saves more cost than ZOM due to its edge-to-edge horizontal federation as it extends the computing capacity and provides the service faster than ZOM. For the edge cost, in the case where offloading takes place based on priority irrespective of job size, the cost of OMNI and E2CVO is lesser than others. The edge cost of E2CVO is better than OMNI, as shown in Fig. 4(b), because of the reverse offloading of some large jobs that have high priority offloaded to the edges. However, in Fig. 5(b), the edge cost of OMNI is lower than E2CVO as the jobs that are offloaded to the edges are smaller, where the offloading decision is taken based on the job size. For the cloud cost, OMNI architecture’s cost is better than E2CVO and similar to others in low input, as shown in Fig. 4(c), where the offloading decision is based on the priority of the jobs. However, the cloud cost of all architectures is nearly similar in the case where the offloading decision is taken based on the job size, as shown in Fig. 5(c).

5.2.2. Total cost analysis – TTSA vs. Others

In Fig. 6, we compared the performance of our proposed TTSA algorithm with two other algorithms: (1) local servers are overloaded (LSAO) used in Mehmeti and Spyropoulos (2016), (2) random (RAND) offloading scheme used in Chen and Hao (2018). In this LSAO scenario, all jobs coming to the edge are directly served by the edges locally, until local servers become overloaded. When the edge becomes overloaded, then tasks are offloaded to the other edges and clouds (Tong et al.,

Table 3

<table>
<thead>
<tr>
<th>Parameter settings</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of clouds</td>
<td>5</td>
</tr>
<tr>
<td>No. of edges</td>
<td>5</td>
</tr>
<tr>
<td>Capacity and bandwidth</td>
<td></td>
</tr>
<tr>
<td>Capacity of an edge server</td>
<td>2 GB</td>
</tr>
<tr>
<td>Capacity of a cloud server</td>
<td>4 GB</td>
</tr>
<tr>
<td>No. of servers in each edge</td>
<td>10</td>
</tr>
<tr>
<td>No. of servers in each cloud</td>
<td>unlimited</td>
</tr>
<tr>
<td>Job size</td>
<td></td>
</tr>
<tr>
<td>small job</td>
<td>job size ≤100 KB</td>
</tr>
<tr>
<td>medium job</td>
<td>100 KB &lt; job size &lt; 25 MB</td>
</tr>
<tr>
<td>Large job</td>
<td>25 MB ≤ job size ≤ 250 MB</td>
</tr>
<tr>
<td>Distance</td>
<td></td>
</tr>
<tr>
<td>User to edge distance</td>
<td>1 KM</td>
</tr>
<tr>
<td>User to cloud distance</td>
<td>1000 KM</td>
</tr>
<tr>
<td>Edge to edge distance</td>
<td>10 KM</td>
</tr>
<tr>
<td>Edge to cloud distance</td>
<td>1000 KM</td>
</tr>
<tr>
<td>Computing cost</td>
<td></td>
</tr>
<tr>
<td>cloud</td>
<td>20 money units/MyCycles</td>
</tr>
<tr>
<td>edge</td>
<td>15 money units/MyCycles</td>
</tr>
<tr>
<td>Communication cost</td>
<td></td>
</tr>
<tr>
<td>Cloud-edge</td>
<td>3 money units/MyBytes</td>
</tr>
<tr>
<td>Edge-edge</td>
<td>2 money units/MyBytes</td>
</tr>
<tr>
<td>User to edge</td>
<td>1 money units/MyBytes</td>
</tr>
<tr>
<td>User to cloud</td>
<td>4 money units/MyBytes</td>
</tr>
</tbody>
</table>

Fig. 4. The total cost of the edge-cloud systems per number of jobs per second in terms of job priority.

Fig. 5. The total cost of the edge-cloud systems per number of jobs per second in terms of job size.
E2CVO due to its reverse offloading architecture. It reverse offloads the jobs for computation. The latency of OMNI is less than the clouds have unlimited computing resources and storage capacity, option to offload to the clouds that have unlimited resources. Since two architectures as they have very negligible waiting time due to the performance of E2CVO and OMNI is much better than the other also do not have the facility to offload them to the clouds. However, the utilization of the edge servers in E2EHO is better than ZOM as an edge can offload the tasks to nearby edges in case of excessive load. The utilization of the cloud servers in E2CVO architecture is better than others, as shown in Fig. 8(a), where offloading takes place based on the job’s priority. Whereas in Fig. 8(b), where offloading is based on the job’s size, cloud servers' utilization is less due to large fragmentation as only large-sized jobs are offloaded from the edges to the clouds. In both cases (priority and size), the edge servers’ utilization in E2EHO is even better than in E2CVO architecture.

5.2.3. Latency analysis – OMNI vs. others

Fig. 7 shows the average latency of the active servers in the edge and cloud layers in both cases, i.e., offloading based on the job’s priority in Fig. 8(a) and offloading based on the job’s size in Fig. 8(b). The utilization of the cloud servers in ZOM and E2EHO are relatively similar in both Fig. 8(a) and (b), as there is no offloading to the cloud in both cases. It only processes the tasks that are received by the cloud from its users. However, the utilization of the edge servers in E2EHO is better than E2CVO architecture. In OMNI architecture, the heavily loaded edges offload their excessive jobs either to clouds vertically or to other edges horizontally. Similarly, the cloud reverse-offload the highly time-sensitive jobs to the edges. The utilization of OMNI architecture is relatively better compared to other architectures both in edge and cloud layer nodes as it tries to accommodate more jobs within the specified amount of resources to maximize resource utilization both in edges and clouds.

5.2.4. Average utilization – OMNI vs. others

Fig. 8 shows the average utilization of the active servers in the edge and cloud layers in both cases, i.e., offloading based on the job’s priority in Fig. 8(a) and offloading based on the job’s size in Fig. 8(b). The utilization of the cloud servers in ZOM and E2EHO are relatively similar in both Fig. 8(a) and (b), as there is no offloading to the cloud in both cases. It only processes the tasks that are received by the cloud from its users. However, the utilization of the edge servers in E2EHO is better than ZOM as an edge can offload the tasks to nearby edges in case of excessive load. The utilization of the cloud servers in E2CVO architecture is better than others, as shown in Fig. 8(a), where offloading takes place based on the job’s priority. Whereas in Fig. 8(b), where offloading is based on the job’s size, cloud servers’ utilization is less due to large fragmentation as only large-sized jobs are offloaded from the edges to the clouds. In both cases (priority and size), the edge servers’ utilization in E2EHO is even better than in E2CVO architecture.

5.2.5. Offloading ratio – number of jobs vs. volume of jobs

Fig. 9 shows the offloading ratio of OMNI architecture in both cases, i.e., based on the job’s priority and size. Fig. 9(a), and (b) present the offloading ratios in terms of the number of jobs, and volume of jobs, respectively. Let us compare the offloading ratios in terms of the number of jobs in Fig. 9(a), in case of priority. The number of jobs offloaded from cloud to edge is only about 5%–6% as the offloading takes place based on the job’s priority, irrespective of the size of the jobs. Whereas in the case of size, the number of jobs offloaded from the cloud to the edge is significantly more, i.e., 17%–18%, as there are smaller size jobs. Similarly, the number of jobs offloaded from the edge to the cloud is only 17%–18% in the case of offloading based on the job’s priority, whereas it is 33%–34% when offloading is based on the job’s size. As Fig. 9(a) shows, when we calculate the offloading ratio in terms of the number of jobs, we found a significant difference in the offloading ratios between the offloading based on the job’s priority and offloading based on the job’s size. However, that is not the case when we calculate the offloading ratios in terms of the volume of the jobs. Fig. 9(b) shows the offloading ratios in OMNI architecture in terms of the volume of the jobs for both cases, i.e., based on the job’s priority and size. As indicated, the offloading ratios in both offloading scenarios are nearly similar.
6. Conclusions

In this paper, we proposed a two-tier federated cloud–edge architecture with omnidirectional offloading. In this federated architecture, the edges can horizontally offload their requests to each other, and not only can edges vertically offload their request to the clouds, but also clouds can reverse offload their highly time-sensitive requests to the edges for faster services. We proposed offloading optimization problem to minimize the total cost of both the cloud and edge layer with given latency constraints, where offloading takes place based on the job’s size and priority. We used a modified SA algorithm to find the global optimum results. The results show our proposed OMNI architecture reduces the total cost by 7%–10%, 20%–30%, and 25%–40% compared to E2CVO, E2EHO, and ZOM architectures, respectively. Our proposed TTSA algorithm also reduces the cost by 45%–55% compared to LSAO and RAND offloading algorithms. It also increases utilization in the edges, and the average latency in our architecture is relatively less compared to other existing architectures.

CRediT authorship contribution statement

Binayak Kar: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – original draft, review & editing, Supervision, Project administration, Funding. Ying-Dar Lin: Conceptualization, Investigation, Writing – review & editing, Supervision, Project administration, Funding. Yuan-Cheng Lai: Conceptualization, Investigation, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article

Acknowledgments

This work was supported by the Ministry of Science and Technology (MOST), Taiwan, under Grant 109-2221-E-011-104-MY3. A preliminary version of this paper appears in the proceedings of IEEE ICC 2020 (Kar et al., 2020). We thank Voel Pater Siswojo, Brandon Wymer Pramana, and Primatar Kuswiradyo for their help with our simulations.

References


