Journal of Computing and Security

Research Article, 2026, Volume 13, Number 1 (pp. 59–71)

http://www.jcomsec.org

Collaborative Fog Computing for Computation Reuse Based on Popularity of Services in IoT

Marzieh Sadat Zahedinia [a,*]

[a] *Faculty of Electrical and Computer Engineering, University of Birjand, Birjand, Iran.*

ARTICLE INFO.

Article history:
Received: 15 October 2025
Revised: 23 May 2026
Accepted: 10 June 2026
Published Online: 02 July 2026

Keywords:
Fog Computing, Computation Reuse, LSH, Network, IoT.

ABSTRACT

Fog computing has emerged as a critical paradigm for processing data closer to the edge. Managing fog nodes effectively is essential for optimizing performance. This study explores a new strategy for reducing response time in IoT, focusing on computation reuse. Computation reuse is a technique used in edge computing to prevent redundant processing. This approach helps in eliminating repetitive calculations, enhancing the capacity of fog computing resources. Although computation reuse saves a significant number of computational resources when the result is cached, it also incurs extra computational costs due to operations like hashing and similarity calculation. It may be beneficial to directly process tasks without searching for the cached results, especially when the probability of successfully finding the target result is low. A novel computation reuse approach among interconnected fog nodes is proposed, leveraging parallel processing and popularity-aware caching to minimize response times and optimize storage. Our method utilizes Locality Sensitive Hashing (LSH) to detect similar computation requests and executes parallel processes for cache lookup and service computation to avoid latency penalties from cache misses. Unlike prior Computation Reuse methods that serialize lookup and execution, the proposed parallel design mitigates cache-miss latency by ensuring that unsuccessful reuse attempts do not increase response time beyond that of direct execution. Evaluations demonstrate significant improvements in response time, reuse rates, energy efficiency, and scalability compared to baseline methods. This work highlights the potential of cooperative computation reuse in fog networks and lays the groundwork for future research in dynamic network topologies.

1 Introduction

The proliferation of the Internet of Things (IoT) has fundamentally transformed how data is generated, processed, and consumed, giving rise to new challenges and opportunities in distributed computing architectures. As IoT continues to integrate into critical domains such as smart traffic control, smart homes, healthcare, and industrial automation, the demand for timely, energy-efficient, and accurate data processing intensifies. Traditional cloud-centric approaches, while powerful, struggle to meet the low-latency and context-aware requirements of many emerging applications, particularly those involving massive volumes

* Corresponding author.
Email address: m.zahedinia@birjand.ac.ir (M. S. Zahedinia)
https://dx.doi.org/10.22108/jcs.2026.147056.1182
ISSN: 2322-4460

of heterogeneous, real-time data [1–3].

To address these limitations, fog and edge computing paradigms have emerged. Fog computing extends cloud computing by bringing computation and storage closer to the devices that generate data, reducing latency and improving efficiency. Fog nodes, which are geographically distributed and resource-constrained, play a crucial role in processing data at the edge [4]. The main purpose of Fog Computing is to reduce the latency and bandwidth consumed by data transferring to the cloud. Because IoT devices have limited resources, they often use the cloud to process and store data. In other words, data management and processing are done by cloud resources. Cloud computing is inappropriate for IoT applications with low latency tolerance. In addition, sending large volumes of IoT data to cloud servers will increase bandwidth consumption [5]. Fog computing mitigates cloud limitations by distributing processing, computing, and storage resources near end-users [6]. This distributed computing architecture positions processing resources between IoT devices and cloud data centers, creating a hierarchical structure that addresses limitations of traditional cloud-based solutions. The growing number of connected devices and increasingly complex applications has highlighted challenges related to scalability, interoperability, real-time response, security, and mobility support within these environments [7].

Three important techniques—service offloading, service caching, and computation reuse—are different from each other. Each of them may be used to improve efficiency.

Service offloading plays a crucial role in enabling devices with limited resources to take advantage of the superior processing capabilities offered by edge and cloud servers as shown in Figure 1. This technological approach has made it possible to support demanding applications like virtual reality and machine learning. However, offloading services introduces its own challenges—most notably, network latency caused by the need to transmit application code, input data, and output results back and forth across the network [2].

To address these issues, caching strategies are widely used. These methods help lower both bandwidth consumption and the total volume of data transferred, while also supporting the stringent latency requirements of time-sensitive applications to enhance user experience. Traditional content caching stores frequently accessed data at the edge, which can significantly decrease response times [8]. Caching mechanisms are optimized to store only selective data, maximizing the effective use of available storage. In contrast, service caching involves keeping service code or virtual machine (VM) images in memory at the edge. The main goal is to reduce both delay and energy consumption linked to network activity, especially when a previously cached service is required again. With service caching, devices that have limited resources can more efficiently offload complex, compute-heavy tasks to a nearby edge server, achieving lower latency than would be possible with a distant cloud server [9].

Computation reuse is another technique where the outputs of certain service computations, along with their respective inputs, are saved for later use. If a future request has the same or similar inputs, these stored results can be returned immediately [10]. This approach can be applied in two main ways:

- **Exact reuse**: Only applies when the new request's inputs completely match those stored.
- **Approximate reuse**: Allows for reuse even if the inputs are similar but not identical.

Computation reuse, which involves caching and reusing intermediate computation results, has been identified as a key strategy to enhance the efficiency of fog computing environments. By minimizing redundant computations, computation reuse can reduce latency, energy consumption, and resource utilization. Computation reuse represents a paradigm that requires storage resources to store previous computations along with efficient indexing and lookup mechanisms. In the context of fog computing, this approach becomes particularly valuable as fog nodes often process similar tasks from multiple IoT devices or end-users. The fundamental principle behind computation reuse is identifying previously executed tasks whose results can be reused for new, similar incoming tasks, thereby avoiding redundant processing.

Connected devices in IoT continuously generate monitoring and measurement data to be delivered to application servers or end-users. Transmitting IoT data through networks would lead to congestion and long delays. In addition, it is necessary to perform processing on this data, and, based on the output, a specific action must be taken. Therefore, in various IoT systems, the need for repetitive computations is often observed [11]. Sensors or IoT devices regularly sense data at specific time intervals. These data indicate a particular parameter in the system. Then, processing is performed on these input data. Finally, the output of the processing is examined, and based on it, an action is taken. If, in consecutive cycles, the data value remains the same, these processes constitute repetitive computations. Therefore, by caching the output of these computations, system performance can be improved.

Despite these advantages, implementing computation reuse in collaborative fog networks is particularly

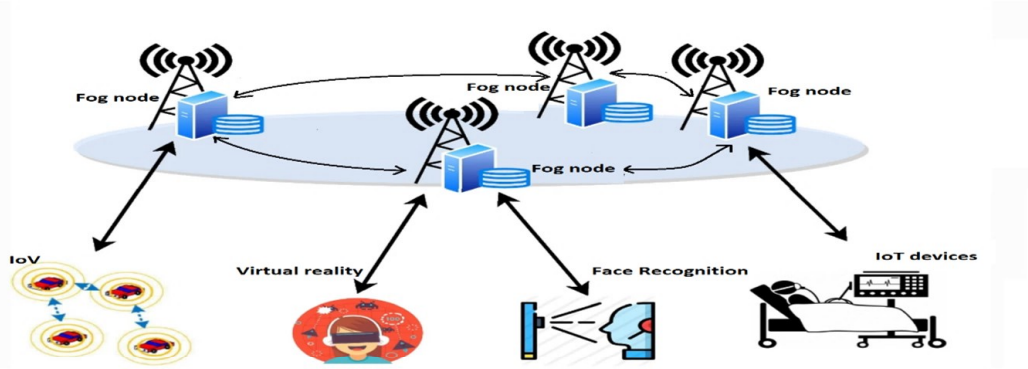

Figure 1. Service Offloading.

challenging for two reasons. First, compared to computation reuse in a single edge server, it is difficult to quantify the mutual effects of storage decisions across multiple edge servers. Second, in contrast to traditional fog networks, introducing computation reuse alters the distribution of task processing times, deeply intertwining and complicating the tasks of storage and scheduling optimization.

While computation reuse has been studied as an effective means to eliminate redundant processing and improve resource efficiency at the edge, existing approaches primarily focus on either single-node reuse (e.g., FoggyCache) or network-level reuse without explicitly addressing the latency penalties caused by cache misses and similarity lookup overhead (e.g., CoxNet and network-based reuse architectures). As a result, the trade-off between reuse benefits and lookup-induced delay remains insufficiently explored, particularly in cooperative fog environments with constrained resources.

This paper addresses this gap by proposing a distributed and cooperative computation reuse framework for interconnected fog nodes that explicitly accounts for the cost of reuse operations. The key novelty of the proposed approach lies in jointly integrating (i) popularity-aware computation caching to selectively store only high-impact computation results, (ii) LSH-based similarity detection to enable scalable reuse of prior computations across neighboring fog nodes, and (iii) a parallel execution strategy in which cache lookup and direct service execution are initiated simultaneously. Unlike prior reuse frameworks that serialize lookup and execution, the proposed parallel design mitigates cache-miss latency by ensuring that unsuccessful reuse attempts do not increase response time beyond that of direct execution.

To control system overhead, two thresholds are employed: a service popularity threshold (th_1) that determines whether reuse mechanisms are activated, and a similarity threshold (th_2) that governs approximate reuse decisions. These thresholds reflect a deliberate design choice to balance reuse benefits against hashing, similarity computation, and coordination overhead, thereby improving robustness under varying workload characteristics.

Our contribution in this paper is threefold:

- This study emphasizes the potential benefits of computation deduplication and reuse in fog computing environments and stresses the importance of developing an approach that accounts for their distributed characteristics—a crucial aspect that past research has largely neglected.
- By presenting a new method to enable the pervasive computation deduplication and reuse at the network based on parallelism, the response time is decreased. Parallelized processing minimizes the negative impact of cache misses. We consider two processes that are run in parallel. The first process involves performing complex computations directly on the edge node, and the second process involves executing complex computations through computation reuse.
- The proposed method considers the popularity of services for computation reuse. To optimize the use of available storage space, only popular services are cached.

The rest of our paper is organized as follows: in Section 2, we give a brief background of the present prior related work. In Section 3, we present the proposed method and IoT use cases that can benefit from computation reuse. In Section 4, we present the evaluation of this method. Finally, in Section 5, we conclude our paper.

2 Related Works

Computation reuse is a fundamental strategy that involves storing intermediate computation results (CRs) and reusing them to avoid redundant computations. Cross-device approximate computation reuse minimizes redundant computation by reusing previously computed outputs with high confidence. This approach is particularly useful in mobile and IoT scenarios where similar applications are invoked on multiple devices in proximity. Techniques such as adaptive locality-sensitive hashing (A-LSH) and homogenized k-nearest neighbors (H-kNN) enable scalable and high-quality reuse opportunities [12].

Approximate computing has emerged as a promising technique to improve performance for many mission-non-critical and error-tolerant applications. The following strategies have been proposed to enhance computation reuse in approximate computing. The ACR framework relaxes the exact matching requirement in inputs to enable computation reuse for approximate computing. This framework uses a similarity quantification scheme to reuse similar results, reducing redundant computations [13].

An auto-memorization processor has been proposed to dynamically detect reusable functions and store their inputs and outputs in a lookup table. By tolerating partial input mismatch, this processor achieves approximate computing with negligible quality deterioration in outputs [14]. While these approaches demonstrate the effectiveness of approximate reuse, they are primarily designed for single-device or tightly coupled execution environments and do not explicitly address cooperative reuse across distributed fog nodes.

FoggyCache is a cross-device approximate computation reuse framework that minimizes redundant computation by reusing previously computed outputs with high confidence. This framework incorporates approximate reuse as a service in the computation offloading runtime, reducing computation latency and energy consumption. However, FoggyCache relies on centralized coordination and does not explicitly consider cooperative reuse among interconnected fog nodes, nor does it address the latency overhead incurred by cache lookup failures in distributed environments.

A network-based computation reuse architecture allows edge servers to store and reuse results from previously executed tasks, significantly reducing task completion time and resource utilization [15, 16].

Matching-based service offloading schemes, such as WHISTLE, facilitate the efficient distribution of tasks to edge servers, achieving a reduction in computation and communication overhead [17]. Although effective in improving offloading efficiency, these approaches primarily focus on task scheduling and placement rather than the joint optimization of reuse decision-making, similarity detection overhead, and cache-miss latency.

CoxNet is an architecture designed to efficiently reuse computations in edge computing environments [18]. CoxNet allows edge servers to leverage previously performed computations while managing the scheduling of incoming dependent tasks. The authors proposed an analytical framework that integrates computation reuse with offloading of dependent tasks and developed an innovative scheduling strategy for computation offloading. The evaluation of this method reveals that it can decrease task execution time with synthetic data and with real-world data. Nevertheless, CoxNet assumes centralized coordination and sequential reuse decision processes, and it does not explicitly address the trade-off between reuse lookup overhead and direct execution latency in highly distributed fog environments.

The following table provides a comparative analysis of the key strategies for managing fog/edge nodes based on computation reuse:

Research Gap and Motivation Despite the substantial progress in computation reuse, existing approaches exhibit three key limitations when applied to interconnected fog computing environments:

(1) reuse decisions are often centralized or local-only, limiting scalability and cooperative gains;
(2) similarity detection overhead is treated as negligible, despite its non-trivial impact on response time under low reuse probability; and
(3) cache lookup and service execution are typically performed sequentially, leading to unnecessary latency when cache misses occur.

To the best of our knowledge, prior work does not adequately address the joint optimization of distributed reuse cooperation, similarity detection overhead, and cache-miss latency mitigation in fog networks. In contrast to FoggyCache, CoxNet, and other network-based reuse methods, this work proposes a fully distributed and cooperative computation reuse framework for interconnected fog nodes that explicitly targets response time minimization. The parallel execution of the proposed model assumes that cache lookup and service execution can be independently scheduled on fog nodes. The overhead of parallel execution is limited to lightweight synchronization and conditional result selection, which is explicitly accounted for in the system model and evaluation. By allowing direct execution to proceed in parallel with reuse lookup, the proposed design avoids the response time penalties typically associated with cache misses, a limitation

Table 1. Summary of Previous Work Methods.

Strategy	**Focus Area**	**Evaluation metric**
Approximate Computation Reuse (ACR)	Relaxing exact matching requirement for computation reuse	Energy Reduction, Classification Accuracy
Auto-Memorization Processor	Dynamic detection of reusable functions and approximate computing	Execution Cycles Reduction
FoggyCache	Cross-device approximate computation reuse	Reuse Rate, Computation Latency Reduction, Energy Consumption
Network-based computation reuse architecture	Architecture of network for computation reuse	Task Completion Time, Computation Correctness
WHISTLE	Efficient distribution of tasks to edge servers	Task Completion Time, Computation Reduction
CoxNet	Architecture of network for computation reuse	Task execution time, Computation Efficiency

not addressed in prior reuse frameworks.

3 Proposed Method

For different network users, the response time is crucial. To reduce the response time, tasks are offloaded from the end devices to the fog or edge nodes. In addition to the offloading mechanism, computation reuse is employed.

We consider two examples of IoT applications and the occurrence of repetitive computations within them. In a smart traffic control scenario, the traffic volume is estimated by identifying the number of car license plates in an image. Cameras at the site capture several snapshots of the environment in very short intervals—just one or a few seconds. These images are, in some cases, very similar to one another. Smart homes are another common IoT application. In smart home applications, device control is based on voice commands received from the homeowner or another household member. In these cases, the computation required by the fog nodes is speech identification and command detection within the person's speech.

In most IoT applications, computations are repetitive. However, computation reuse (CR) can only be employed in two cases: first, when similar computations are offloaded to the same fog node, and second, when fog nodes are connected and can use each other's previously computed results.

Recently, extensive research has been conducted on computation reuse in a single edge server [12, 13, 17]. However, the benefits of computation reuse can only be fully realized through the cooperation of multiple edge servers. The main reason is that, through cooperation, cached computation results on one edge server can be reused by tasks offloaded to other edge servers. This not only improves the cache hit rate for each stored result but also increases the total number of results stored in the system, since it eliminates the need for redundant storage of similar results on different edge servers.

Computation reuse is an important concept in computer science and engineering, particularly in the fields of high-performance computing, machine learning, and distributed systems. It involves reusing previously computed results to save time and resources, especially when the same computations are needed multiple times.

Therefore, it is necessary to cache records for some previously executed computations. Each record includes the input data and the output of relevant computations. If these services are offloaded again with different input data, a lookup is first performed. If the service is for the same data or data that is sufficiently similar, the cached output is returned. Otherwise, if the search is unsuccessful, the computations will be executed directly on the fog node.

In most studies, computation reuse has only been considered within a single fog node, and less attention has been given to the connection between neighboring fog nodes. However, the benefits of computation reuse can be fully realized only through the cooperation of multiple fog servers. The main reason is that, with cooperation, the computation results stored in one fog server can be shared by tasks offloaded to other edge servers. This not only improves the cache hit rate, but it also increases the total number of results stored in the system, as there is no longer a need to redundantly store the same results on multiple fog

servers. Specifically, on fog nodes that are near each other, the likelihood of similar computations being needed is higher. Many users in the same geographic area request similar services. Therefore, cooperation between neighboring fog nodes helps improve system performance.

One of the challenges in the problem of computation reuse is the increased response time in cases where the lookup for similar computations is unsuccessful. An unsuccessful lookup leads to additional time.

As emerging applications require extremely fast response times—for instance, augmented reality may need a response in less than 10 milliseconds—removing redundant computations and leveraging previously obtained results can speed up the processing of services delegated by user devices. By drawing on outputs from prior, comparable tasks, systems can avoid repeating costly computations. This method also makes better use of often limited edge computing resources by exchanging storage for computational effort, meaning it conserves and reapplies earlier results. Ultimately, computation reuse has the potential to greatly boost both the efficiency and capacity of edge computing systems.

To address the mentioned challenge, we explore and evaluate a new approach. The proposed method has three main features: (1) Fog nodes are interconnected; (2) The popularity of services is considered when storing computations; and (3) The fog server utilizes a parallel technique. For this purpose, we consider two processes, both of which start simultaneously. Process one: direct execution of the service and determination of its output. Process two: obtaining the service output by reusing prior computations. By default, the cache lookup and its related operations are completed first. Then, if the search result is unsuccessful, the computation for the requested service is started. The first feature is intended to improve the efficiency of computation reuse, and the second feature aims to enhance the efficiency of service caching within the network. The third feature also aims to reduce response time.

When a fog node receives a complex computational service, the fog server operates in a parallel manner—that is, it simultaneously executes the offloaded service and performs a lookup among previously cached computations. In this way, if the search fails (meaning the service and its input data do not have a sufficiently high similarity), no time is lost, as both processes are carried out simultaneously on the fog server. On the other hand, if the search for duplicate computations for a request fails, a neighboring node is asked to search for duplicates among its own computations. Thus, fog nodes are coordinated with their neighboring nodes. Since similar computations might be offloaded to different servers, this feature helps improve the system's overall performance. If the search is unsuccessful, the input data and its results are cached. The algorithm's operation in different scenarios is described in Section 3.3. Figure 2 shows the flowchart of the proposed method. According to this flowchart, three different paths exist for terminating the algorithm depending on the simulation scenario. When a service request arrives at a fog node, the decision process is initiated. In this stage, the service popularity is calculated based on the number of previous requests within a specified time window. The computed popularity value is then compared with a predefined threshold (th_1). If the service is considered non-popular, the system proceeds directly with normal service execution. In this case, the computation reuse mechanism is not activated in order to avoid unnecessary lookup overhead. However, if the service is identified as popular, the system switches to a logical parallel processing mode, where two processes are initiated simultaneously.

3.1 System Model and Problem Formulation

In this section, we describe the proposed method and provide its corresponding formulation. We consider a network with N fog nodes indexed by $N = \{1, 2, \ldots, N\}$. Each IoT device is assigned to a fog node. Each fog node has a computational capacity and storage capacity.

Let's assume that there exists a set of M different services $M = \{c_1, c_2, \ldots, c_M\}$ which are requested at rates $R_i = \{R_1, R_2, \ldots, R_M\}$ by a set of users. Each of these services has a different request rate. Depending on the specific application, some services are more popular and are requested by more users. In contrast, some services are less popular.

Users utilizing these services might offload computation tasks to nearby fog nodes, each with different input data. Based on the findings in [2], for a specific application, tasks with input data that are sufficiently similar tend to produce identical outputs. Therefore, the output of earlier tasks can be reused for later similar tasks.

Specifically, for service $x \in M$, we have assumed a set of typical inputs $D_x = \{d_1, d_2, \ldots, d_m\}$ from its historical data. For each service x, one of the data items from D_x can be requested as input data by any end user.

Optimization Objective: The primary goal is to minimize the average response time for user requests across the fog network. This is achieved by intelligently leveraging cached computational results to bypass redundant execution, balanced against the overhead

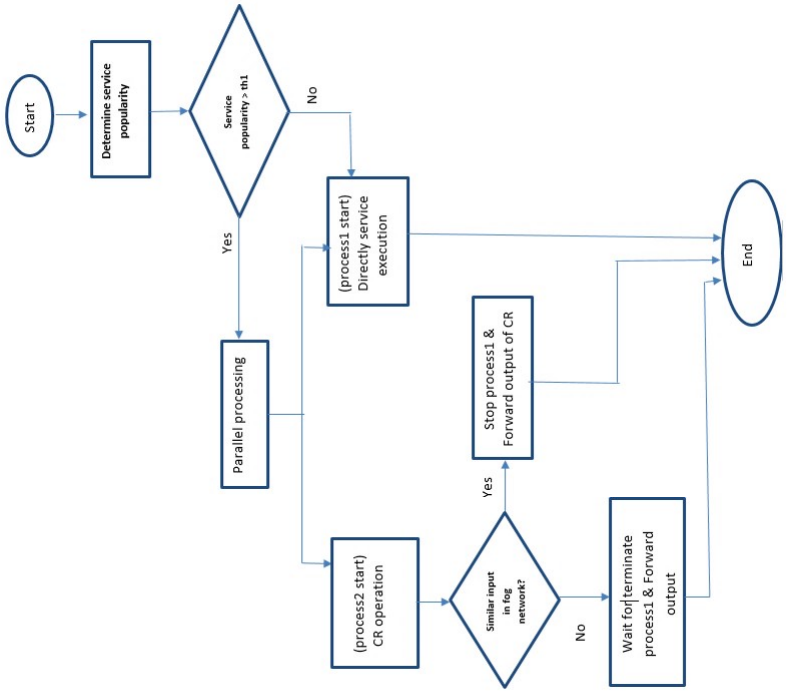

Figure 2. Flowchart of the Proposed Method.

of cache management and similarity search.

Decision Framework and Threshold Rationale: To manage the trade-off between the benefits of reuse and its inherent overheads, we employ a threshold-based decision policy executed upon a request arrival for service x with input data d_i:

Popularity Filter (Threshold th_1): The system first evaluates the service's popularity, $PO(i)$, defined as the proportion of all requests at the node for service i (Section 3.2). A cache lookup is initiated *only if* $PO(i) > th_1$. The overhead of similarity search (via Locality Sensitive Hashing, Section 3.3) is justifiable only for services requested with sufficient frequency. For low-popularity services, the probability of a cache hit is low, and the resource cost of maintaining their cache entries often outweighs the occasional benefit.

Similarity Check (Threshold th_2): If a service is popular ($PO(i) > th_1$), a parallel process is launched to search for cached data d' similar to the new input d using LSH. The similarity $\text{sim}(d, d')$ is computed [10]. A cached result is reused *only if* $\text{sim}(d, d') \geq th_2$. This threshold enforces the correctness and utility of reuse. A low th_2 increases reuse frequency but risks returning incorrect outputs for insufficiently similar data, compromising application integrity. A high th_2 ensures high output fidelity but reduces reuse opportunities. The optimal th_2 is application-dependent and should be calibrated based on the sensitivity of the service's output to input variations.

Concurrently with the cache lookup process, a second process executes the service natively. The system returns the first available valid result (from either process) and terminates the other, minimizing response time.

The proposed approach assumes logical parallelism modeled at the system level, rather than idealized zero-cost hardware-level parallelism. In practice, fog nodes are multi-core computing units capable of handling concurrent threads. Therefore, parallel execution is implemented as two independent threads scheduled by the local operating system scheduler.

3.2 Service Popularity

The popularity of a service is measured by how frequently it is requested. Each time a fog node receives a request for a specific service, that service's popularity increases. Let the total number of requests arriving at a fog node for service i; the popularity of that service is calculated as the proportion of its requests to the overall requests for all services. A service that receives more requests is considered more popular, and such services are labeled as popular services. The popularity of a service i can be expressed as:

$$PO(i) = \frac{R_i}{R_{total}} \tag{1}$$

where

$$R_{total} = \sum_{j=1}^{N} R_j \tag{2}$$

and R_i and R_{total} are the number of received requests for service i to the fog node and the total number of received requests for all services, respectively, and N is the total number of available services in the network.

3.3 Similarity Measurement

The similarity between two inputs d and d' is computed using a fixed, service-specific similarity function $\text{sim}(d, d')$. The similarity function $\text{sim}(d, d')$ is normalized and bounded in the range $[0, 1]$, where 0 denotes no similarity and 1 denotes maximum similarity. This normalized similarity value is used consistently for cache indexing, similarity evaluation, and reuse decisions.

Higher values indicate greater similarity. This similarity function is consistently used for cache indexing, similarity evaluation, and reuse decisions. Reuse is allowed if $\text{sim}(d, d') \geq th_2$. The specific form of $\text{sim}(d, d')$ is application-dependent but fixed per service type (e.g., cosine similarity, normalized Euclidean similarity, or domain-specific similarity functions). Cosine similarity is suitable for high-dimensional embedding-based data where vector direction is more meaningful than magnitude (e.g., text, image, and semantic embeddings). Normalized Euclidean similarity is appropriate for real-valued numerical data where absolute distances are semantically meaningful (e.g., IoT sensor data and physical measurements). Domain-specific similarity functions are used for structured or complex data types (e.g., images, videos, graphs, or industrial processes), where similarity depends on application-specific semantics rather than purely numerical distance. Therefore, depending on the data and the intended application, one of these algorithms can be used.

3.4 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) is an algorithmic technique used to efficiently find similar or nearby data items in high-dimensional spaces. It works by hashing input items so that similar items have a high probability of being mapped into the same bucket(s) of a hash table. This allows for fast approximate nearest neighbor searches because instead of comparing every item against all others, only items in the same buckets are compared. Unlike traditional hashing, which aims to minimize collisions, LSH intentionally maximizes collisions for similar items to enhance similarity detection. LSH is used to find the k-nearest neighbors of an incoming item i by applying a hash function h to i and using the resulting hash value $h(i)$ to identify the bucket(s) in a hash table where the nearest neighbors may be located. To enhance the search accuracy, especially for high-dimensional data types such as images or videos, multiple hash functions $h_1, h_2, \ldots, h_n$ can be applied to each incoming item, with each hash function indexing a different hash table.

LSH is used as a candidate retrieval mechanism, not as the final similarity decision rule. For each service x, the fog node maintains L independent hash tables $HT_1, HT_2, \ldots, HT_L$. Each hash table HT_l is associated with a hash function h_l. These hash functions satisfy the LSH property that inputs which are similar under the similarity metric $\text{sim}(d, d')$ have a high probability of being mapped into the same hash bucket. For each cached input d', indexing is performed as follows:

$$d' \rightarrow \{h_1(d'), h_2(d'), \ldots, h_L(d')\} \tag{3}$$

The input is stored in the corresponding buckets of all L hash tables.

For a newly arriving input d of service x:

(1) Hashing is performed using all hash functions $\{h_1(d), h_2(d), \ldots, h_L(d)\}$.
(2) Candidate inputs, $C(d)$, are retrieved from the corresponding hash buckets. $C(d)$ contains only those previously stored inputs that are likely to be similar to d according to the LSH hashing stage. It is a reduced search space, meaning the system does not compare d with all cached data, but only with the candidates in $C(d)$.
(3) Exact similarity is computed only for candidate inputs. For each d' in $C(d)$ compute $\text{sim}(d, d')$.
(4) If there exists d' in $C(d)$ such that $\text{sim}(d, d') \geq th_2$, the cached result is reused.

Thus, LSH only reduces the search space, while the final reuse decision is based on deterministic similarity evaluation.

3.5 Service execution time

In the proposed method, the response time is estimated based on different scenarios. Two threshold values are used in this method. th_1 represents the popularity threshold of the service, and th_2 represents the similarity threshold of the input data with cached data. Only services with popularity higher than th_1 are stored for reuse. For service i, if R_i exceeds the threshold, the service output may be returned from previous computation results. If R_i does not meet the threshold, the search and similarity detection operations are not performed. In this case, the output is determined by directly performing the computations. If we denote the execution time of service i with input data j by $t_{i,j}$ and the execution times of process one and process two by $t_{i,j}^{proc1}$ and $t_{i,j}^{proc2}$ respectively, based on the explanations provided, we have:

$$t_{i,j} = \begin{cases} \min\left(t_{i,j}^{proc1}, t_{i,j}^{proc2}\right) & R_i > th_1 \\ t_{i,j}^{proc1} & R_i \leq th_1 \end{cases} \quad (4)$$

To clarify the proposed method, we consider three different scenarios. In the scenarios considered, we assume that the users are interested in specific services and that the users' devices lack the processing power to perform the desired service. Therefore, they offload these to the fog nodes.

(a) In the first scenario, after offloading the service to the fog node, the popularity of the service is first calculated at the fog node. In this scenario, we assume that the requested service's popularity is less than th_1. Therefore, the requested service is executed directly by the fog node, and the result is returned to the user.

(b) In the second scenario, after offloading the service to the fog node, the popularity of the service is first calculated at the fog node. It is assumed that the popularity of the requested service is higher than the threshold th_1. Based on this assumption, there is a possibility that this service and similar input data are available in the cache. Process 1, which is the direct execution of the service, begins. In parallel with this process, Process 2 is carried out, which involves searching the cache of the fog node and its neighbors. If the cache search is successful, the output is returned from previous calculations. The data similarity is greater than the threshold (th_2), so the response time is less than in Scenario 1.

(c) The third scenario is similar to scenario b. We assume that the popularity of the requested service is greater than the threshold th_1. However, data similar to the input data is not in the cache; the data similarity is less than the th_2. Process 1 and Process 2 start simultaneously, but because data similar to the input is not in the cache, the output is determined via Process 1. The response time is equal to that in Scenario 1.

4 Evaluation

This evaluation section provides a comprehensive analysis of the proposed method. The evaluation is structured as follows. First, we delineate the experimental design and performance metrics. Next, we present both simulation results, comparing the proposed approach against baseline methods and analyzing its impact on response time, cache efficiency, energy consumption, and system scalability. The proposed method was compared against several approaches. The first approach is **Local-only Computation Reuse.** Computation reuse is enabled solely within individual fog nodes, with no inter-node cooperation. The second approach is **No Computation Reuse.** Each service is processed independently, with no caching or reuse, representing the traditional fog execution model. Finally, the third one is **Centralized Caching.** Cached results are maintained at a centralized location, requiring data transmission from edge devices to the core for lookup and retrieval.

For evaluation, three metrics are considered:

- **Average Response Time:** The mean time from task arrival to result delivery, reflecting system latency. This metric directly captures end-to-end system latency and is critical for latency-sensitive applications such as augmented reality, smart transportation, and real-time analytics.
- **Percentage of reuse:** The proportion of requests served from computation reuse, indicating the effectiveness of computation reuse. This metric quantifies the effectiveness of computation reuse and reflects the system's ability to exploit redundancy in service requests and input data.
- **Scalability:** Evaluated by observing performance trends as the number of services increases, reflecting the system's ability to maintain efficiency under growing service diversity and cache pressure.

We evaluated the proposed method using the NS3 simulator. We assume a network consisting of several users, intermediate routers, and edge nodes. There are 50 users, each connected to an edge node with two or at most three hops. The number of edge nodes is 5. Edge nodes are server-type nodes that have processing power and storage capacity and randomly provide some services. The total number of defined services is 100. The user offloads the requested service and input data to the edge node. Users can request any of the 100 services. For services whose popularity is above the threshold, computation reuse may be used. If the service is not available on the corresponding edge node, the service request is sent to the neighboring edge nodes.

The rate of user request submissions is between 3 to 10 services per second. The number of data items defined for each service is randomly between 4 and 10. The execution time of services directly (by the edge node) ranges from 20 ms to 50 ms. With the increase in the similarity threshold, the number of similar services decreases. Therefore, the possibility of obtaining computation results through the reuse of calculations is reduced, and ultimately, the response time at a 90% threshold is close to the other two methods, as shown in Figure 3.

All reported results represent averaged values over

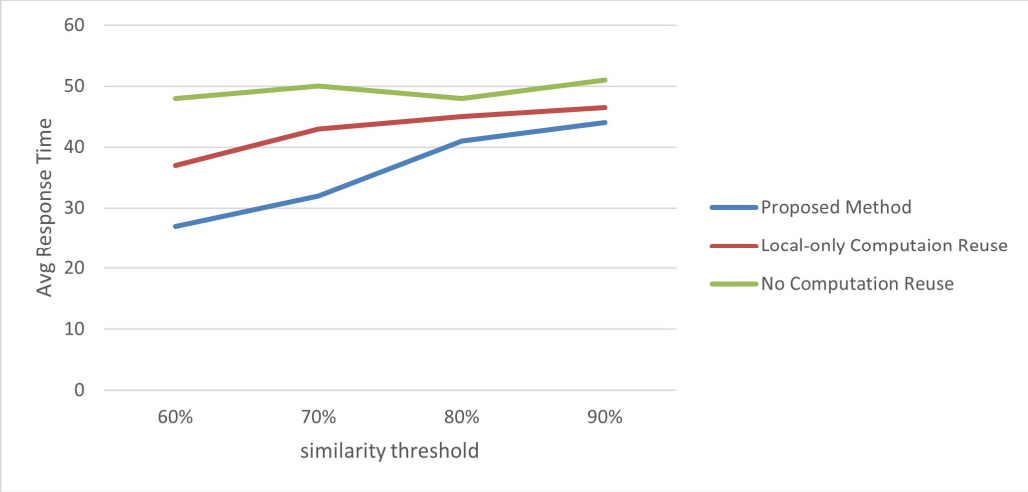

Figure 3. Average Response Time vs Similarity Threshold.

5 independent simulation runs. This ensures that reported performance gains are not artifacts of stochastic simulation variability.

By keeping the similarity threshold constant at 60% and increasing the number of services available in the network, we observe that the response times in different methods become closer to each other. This reflects increased service diversity and reduced reuse probability. Nevertheless, the proposed method consistently maintains lower response times due to cooperative caching and parallel execution, demonstrating robustness under increasing system scale, as shown in Figure 4.

In the No Computation Reuse approach, each service is processed independently. This means that no computations are cached in advance, and no response is received from cache or neighboring nodes. As a result, all processing steps must be performed from scratch. This approach consumes the most processing power. In the central caching approach, the computational results are cached in a central location. Although this approach requires sending services to the center, the processing load caused by the computations is reduced due to the extensive use of previous results. The proposed method significantly reduces the processing load by enabling parallel execution. This method performs better, especially in scenarios with popular services. This is a fact that is also evident from the simulation results, as shown in Figure 5. Since CPU utilization is used as a proxy for energy consumption, these results indicate improved energy efficiency through reduced redundant computation.

To examine scalability, we also examine the increase in the number of user devices. The results are shown in Figure 6. As the table shows, with the increase in the number of users and the similarity threshold remaining constant at 60% and other simulation parameters remaining constant, the average response time increases in all three methods. With the increase in the number of users, although the variety of requested services increases, the reuse of calculations can still be useful for improving the average response time.

By enabling parallel execution of cache lookup and computation, the method ensures that unsuccessful lookups do not increase latency, a critical requirement for real-time applications such as augmented reality and smart traffic management. Inter-node cooperation increases the likelihood that a requested computation result is already available somewhere in the network, especially for popular services. Popularity-aware caching ensures that limited storage resources are allocated to computations with the highest expected reuse, maximizing cache efficiency. By reusing computation results and reducing redundant processing, the method conserves energy, a key concern in edge/fog environments.

Despite its strengths, the proposed method faces several challenges. Synchronizing caches across distributed fog nodes introduces additional complexity, particularly in highly dynamic environments. Querying neighboring nodes for cached results incurs communication overhead, which must be balanced against the potential gains in cache hit rate. The effectiveness of LSH and similarity detection depends on the choice of metric and parameters, which may require domain-specific tuning. As the number of users and services grows, maintaining efficient cache lookup and update operations becomes increasingly challenging. In real fog networks, due to the parallel execution of two processes, we will have overhead. This overhead will be in terms of using processing resources to perform service calculations and similarity search. In terms of memory, the proposed method also requires storing the hash ta-

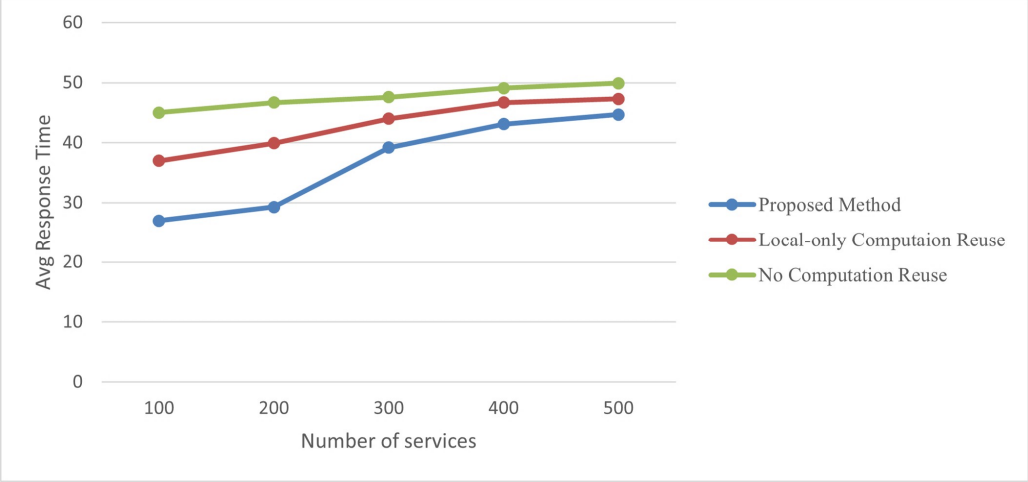

Figure 4. Average Response Time vs Number of Services.

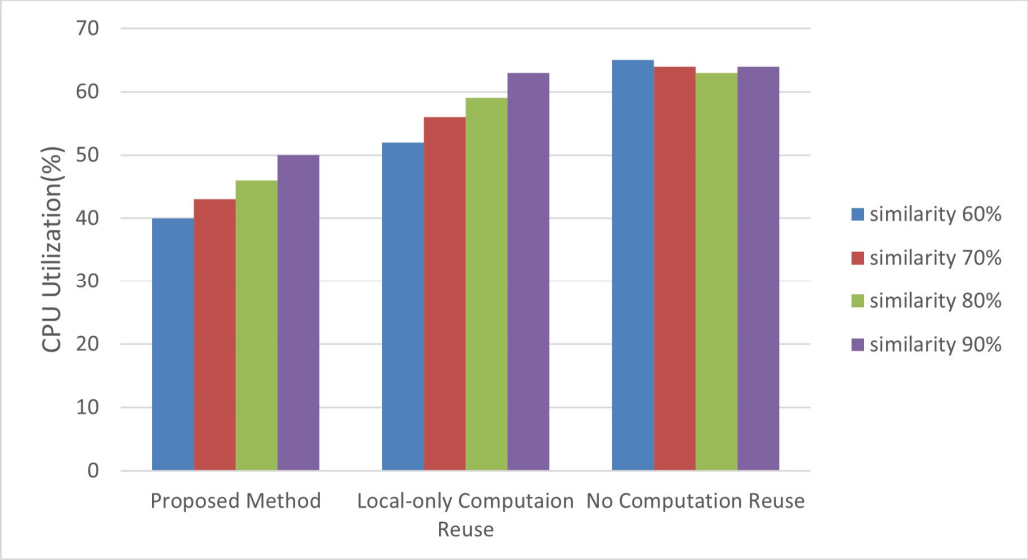

Figure 5. Percentage of CPU Utilization.

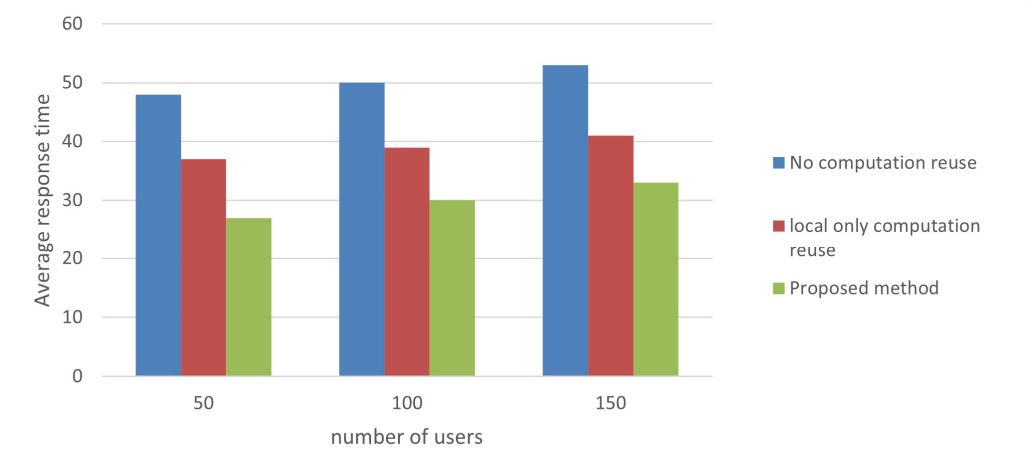

Figure 6. Average Response Time vs Number of Users.

ble as well as the input data and output results of the calculations. In addition, parallel execution requires synchronization and time management of resources. This synchronization will lead to overhead.

5 Conclusions and Future Directions

The effective management of fog nodes based on computation reuse is crucial for optimizing resource utilization and performance in fog computing environments. Various strategies, such as caching and computation reuse, resource allocation, task scheduling, approximate computing, and learning-based approaches, have been proposed to enhance the efficiency of networks. These strategies have been shown to reduce latency, energy consumption, and resource utilization while meeting the QoS requirements of applications.

In this paper, an adaptive, cooperative method for computation reuse in fog computing environments for IoT applications is proposed. The evaluation has demonstrated that the proposed parallel and cooperative computation reuse method delivers substantial benefits for facilitated fog computing in IoT environments. By enabling fog nodes to share cached results, targeting storage towards popular services, and executing cache lookup and computation in parallel, the method achieves significant reductions in response time and energy consumption, while improving cache hit rates, storage efficiency, and system throughput.

Future works will focus on extending the method to support dynamic network topologies, integrating advanced prediction and prefetching mechanisms, and empirically validating performance in operational IoT testbeds. By systematically addressing the challenges of computation reuse in distributed fog environments, this research contributes a foundational building block for the next generation of efficient, responsive, and intelligent edge computing systems. Future works can dynamically adjust thresholds based on observed workload characteristics, further optimizing performance.

References

[1] A. Koohang, C. S. Sargent, J. H. Nord, and J. Paliszkiewicz, "Internet of Things (IoT): From awareness to continued use," *Int. J. Inf. Manage.*, vol. 62, p. 102442, Feb. 2022, doi: 10.1016/j.ijinfomgt.2021.102442.

[2] M. W. Al Azad and S. Mastorakis, "The Promise and Challenges of Computation Deduplication and Reuse at the Network Edge," *IEEE Wireless. Commun.*, no. December, pp. 112–118, 2022, doi: 10.1109/MWC.010.2100575.

[3] M. S. Zahedinia, "Integration of fog computing in NDN-based IoT," J. Soft Comput. Inf. Technol., vol. 14, no. 3, pp. 51–60, 2025, doi: 10.22034/jscit.2025.540836.2157.

[4] T. P. Da Silva *et al.*, "Fog Computing Platforms for Smart City Applications: A Survey," *ACM Trans. Internet Technol.*, vol. 22, no. 4, 2022, doi: 10.1145/3488585.

[5] M. S. Zahedinia, M. R. Khayyambashi, and A. Bohlooli, "Fog-based caching mechanism for IoT data in information centric network using prioritization," *Comput. Networks*, vol. 213, no. March 2021, p. 109082, 2022, doi: 10.1016/j.comnet.2022.109082.

[6] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and Internet of Things: A survey," Future Gener. Comput. Syst., vol. 56, pp. 684–700, Mar. 2016, doi: 10.1016/j.future.2015.09.021.

[7] H. Sabireen and V. Neelanarayanan, "A Review on Fog Computing: Architecture, Fog with IoT, Algorithms and Research Challenges," *ICT Express*, vol. 7, no. 2, 2021, doi: 10.1016/j.icte.2021.05.004.

[8] M. S. Zahedinia, M. R. Khayyambashi, and A. Bohlooli, "IoT data management for caching performance improvement in NDN," *Cluster Comput.*, 2023, doi: 10.1007/s10586-023-04197-2.

[9] M. Aazam, S. Zeadally, and K. A. Harras, "Offloading in fog computing for IoT: Review, enabling technologies, and research opportunities," *Futur. Gener. Comput. Syst.*, vol. 87, pp. 278–289, 2018, doi: 10.1016/j.future.2018.04.057.

[10] A. Javaheri, A. Bohlooli, and K. Jamshidi, "Enhancing computation reuse efficiency in ICN-based edge computing by modifying content store table structure," *Computing*, vol. 106, pp. 2949–2969, 2024, doi: 10.1007/s00607-024-01312-y.

[11] A. Javaheri, A. Bohlooli, and K. Jamshidi, "Boosting computation reuse efficiency in ICN-based edge computing via improved forwarding algorithms," *Computing*, vol. 107, p. 137, 2025, doi: 10.1007/s00607-025-01489-w.

[12] P. Guo, B. Hu, R. Li, and W. Hu, "FoggyCache," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, New York, NY, USA: ACM, Oct. 2018, pp. 19–34. doi: 10.1145/3241539.3241557.

[13] X. He, S. Jiang, W. Lu, G. Yan, Y. Han, and X. Li, "Exploiting the Potential of Computation Reuse Through Approximate Computing," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 3, no. 3, pp. 152–165, Jul. 2017, doi: 10.1109/TMSCS.2016.2617343.

[14] Y. Sato, T. Tsumura, T. Tsumura, and Y. Nakashima, "An Approximate Computing Stack Based on Computation Reuse," in *Proceedings*

- *2015 3rd International Symposium on Computing and Networking, CANDAR 2015*, Institute of Electrical and Electronics Engineers Inc., Mar. 2016, pp. 378–384. doi: 10.1109/CANDAR.2015.35.

[15] M. W. Al Azad and S. Mastorakis, "Reservoir: Named Data for Pervasive Computation Reuse at the Network Edge," 2022 IEEE International Conference on Pervasive Computing and Communications, PerCom 2022, pp. 141–151, 2022, doi: 10.1109/PerCom53586.2022.9762397.

[16] B. Nour, S. Cherkaoui, and Z. Mlika, "Federated Learning and Proactive Computation Reuse at the Edge of Smart Homes," *IEEE Trans. Netw. Sci. Eng.*, vol. 9, no. 5, pp. 3045–3056, 2022, doi: 10.1109/TNSE.2021.3131246.

[17] B. Nour and S. Cherkaoui, "Matching-based Service Offloading for Compute-less Driven IoT Networks," *Proc. - IEEE Glob. Commun. Conf. GLOBECOM*, pp. 3122–3127, 2024, doi: 10.1109/GLOBECOM52923.2024.10901209.

[18] Z. Bellal, B. Nour, and S. Mastorakis, "CoxNet: A Computation Reuse Architecture at the Edge," *IEEE Trans. Green Commun. Netw.*, vol. 5, no. 2, pp. 765–777, 2021, doi: 10.1109/TGCN.2021.3071497.

Marzieh Sadat Zahedinia received her B.Sc. and M.Sc. degrees from the Iran University of Science and Technology (IUST) and University of Birjand, Iran, in 2010 and 2013, respectively. Subsequently, she received her Ph.D. degree in computer engineering from the University of Isfahan, Iran, in 2023. Presently, she is an assistant professor in the Faculty of Electrical and Computer Engineering, University of Birjand, Birjand, Iran. Her research areas include Computer Networking, Information-Centric Network, Internet of Things (IoT) and caching mechanisms.