

An  
Industry oriented Mini Project Report on  
**PLACEMENT POLICY IN FOG COMPUTING**

A report submitted in partial fulfillment of the requirements for the award of degree of  
Bachelor of Technology

By

**Alla Sai Manideep Reddy**  
**(20EG105402)**

**Beerreddy Harshitha**  
**(20EG105405)**

**Adithya Krishnamurthy**  
**(20EG105417)**



Under the guidance of

**Dr. Pallam Ravi**  
**Assistant Professor**  
**Department of CSE**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**  
**ANURAG UNIVERSITY**  
**VENTAKAPUR-500088**  
**TELANGANA**  
**Year 2023-2024**

## DECLARATION

We hereby declare that the Report entitled **Placement Policy in Fog Computing** submitted for the award of Bachelor of Technology Degree is our original work and the Report has not formed the basis for the award of any degree, diploma, associate ship or fellowship of similar other titles. It has not been submitted to any other University or Institution for the award of any degree or diploma.

Alla Sai Manideep Reddy  
(20EG105402)

Beerddy Harshitha  
(20EG105405)

Adithya Krishnamurthy  
(20EG105417)

## **CERTIFICATE**

This is to certify that the Report entitled **Placement Policy in Fog Computing** that is being submitted by **A. Sai Manideep** bearing Hall Ticket number **20EG105402**, **B. Harshitha** bearing the Hall Ticket number **20EG105405**, **K. Adithya** bearing the Hall Ticket number **20EG105417** in partial fulfillment for the award of **Bachelor of Technology in Computer Science and Engineering** to the **Anurag University** is a record of bonafide work carried out by them under my guidance and supervision.

The result embodied in this Report have not been submitted to any other University or Institute for the award of any degree or diploma.

**Signature of Supervisor**

Dr. Pallam Ravi

Assistant Professor, CSE

**Dean, CSE**

**External Examiner(s)**

## ACKNOWLEDGMENT

We would like to express our sincere thanks and deep sense of gratitude to project supervisor and co-ordinator **Dr. Pallam Ravi, Assistant Professor, Department of CSE** for his constant encouragement and inspiring guidance without which this project could not have been completed. His critical reviews and constructive comments improved our grasp of the subject and steered to the fruitful completion of the work. His patience, guidance and encouragement made this project possible.

We would like to acknowledge our sincere gratitude for the support extended by **Dr. G. Vishnu Murthy**, Dean, Dept. of CSE, Anurag University. We also express my deep sense of gratitude to **Dr. V V S S Balaram**, Academic coordinator, and Project review committee members, whose research expertise and commitment to the highest standards continuously motivated me during the crucial stages our project work.

We would like to express our special thanks to **Dr. V. Vijaya Kumar**, Dean School of Engineering, Anurag University, for their encouragement and timely support in our B. Tech program.

Alla Sai Manideep Reddy  
(20EG105402)

Beerreddy Harshitha  
(20EG105405)

Adithya Krishnamurthy  
(20EG105417)

## ABSTRACT

The Internet of Things (IoT) has the overarching goal of connecting a wide array of objects, including smart cameras, wearable devices, environmental sensors, home appliances, and vehicles, to the internet. This connectivity generates a massive volume of data that can strain storage systems and data analytics applications. While cloud computing offers scalable infrastructure services to accommodate IoT storage and processing demands, there are applications like health monitoring and emergency response that require low latency. The delay introduced by transferring data to the cloud and back can significantly hinder the performance of such applications. To address this limitation, the concept of Fog computing has been introduced. In the Fog computing paradigm, cloud services are extended to the edge of the network, reducing latency and network congestion. This approach aims to realize the full potential of both Fog and IoT paradigms for real-time analytics. However, to make this a reality, several challenges must be addressed.

The most critical challenge is the design of resource management techniques. These techniques determine which modules of analytics applications should be pushed to each edge device to minimize latency and maximize throughput. To tackle this challenge effectively, we proposed the JAYA algorithm which is specialized for optimizing the placement of application modules. It is essential to have an evaluation platform that allows for the quantification of the performance of resource management policies on an IoT or Fog computing infrastructure in a repeatable and systematic manner. In this project we used a simulator, called iFogSim, to model IoT and Fog environments and measure the impact of resource management techniques in latency, network congestion, energy consumption, and cost.

We present the results of this experimentation and analysis, demonstrating the effectiveness of the Jaya algorithm in dynamically optimizing resource allocation in Fog computing. Through a comparative evaluation with traditional allocation policies, we showcase notable reductions in latency, energy consumption and improvements in data processing efficiency. The findings reveal the potential of the Jaya algorithm in enhancing the real-time analytics capabilities of Fog computing environments, thereby addressing critical challenges in IoT applications such as health monitoring and emergency response.

## **LIST OF FIGURES**

Figure. No.	Figure. Name	Page No.
Figure 1.1	Cloud service, Iot, Fog Environment	2
Figure 1.1.1	Distributed data processing in Fog Computing	3
Figure 1.2.1	Fog Architecture	5
Figure 1.3.1	High-Level view of interactions among Fog iFogSim components	6
Figure 3.2.1	Flowchart of JAYA Algorithm	14
Figure 3.3.2.	Initial Population Matrix	15
Figure 4.3.1	ModulePlacementOnlyCloud.java file	23
Figure 4.3.2	VRGameFog.java file	23
Figure 4.3.3.	Output File	24
Figure 4.4.1	Eclipse IDLE	26
Figure 5.1.1.	Execution Times of each algorithm	28
Figure 5.2.1	Energy Consumed by cloud	29

## **LIST OF TABLES**

Table No.	Table Name	Page No.
Table 1.4.1.	Latency Sum Problem illustration	8
Table 2.4.1.	Comparison of Literature	10-11
Table 3.3.1.	Evaluation of Fitness Values	15
Table 3.3.2	Best and Worst Solutions are identified	16
Table 3.3.3.	Best and Worst Values	16

Table 3.3.4	Random values for $x_1$	16
Table 3.3.5	Random vales for $x_2$	16
Table 3.3.6	$X_{\text{new}}$ values	17
Table 3.3.7	Updated values of $F(x)$	18
Table 5.1.1	Latencies of different algorithms	28
Table 5.2.1	Energies consumed by cloud in different algorithms	29



## INDEX

<b>S.No.</b>	<b>CONTENT</b>	<b>Page No.</b>
1.	Introduction	1
	1.1 Fog Computing – Definition and concept	3
	1.2 Fog Computing Architecture	4
	1.3 Fog Computing Framework	6
	1.3.1 Physical Components	6
	1.3.2 Logical Components	7
	1.3.3 Management Components	7
	1.4 Placement Policies Problem illustration	8
2.	Literature	9
	2.1 Particle Swarm Optimization (PSO)	9
	2.2 Enhanced Particle Swarm Optimization (EPSO)	9
	2.3 Cloud- Only Algorithm	10
	2.4 Comparison of Literature	10
3.	Placement Policies Of Fog Nodes	11
	3.1 JAYA algorithm	11
	3.2 JAYA algorithm steps	12
	3.3 Placement Policy illustration	15
	in JAYA algorithm	

4.	Implementation	19
	4.1 Functionalities	19
	4.2 Attributes	20
	4.3 Environment Screenshots	23
	4.4 Experimental Setup	24
	4.5 Parameters	26
	4.5.1 Latency	26
	4.5.2 Energy Consumption	27
5.	Discussion of Result	27
	5.1. Latency	27
	5.2 Energy Consumption	28
6.	Conclusion	30
7.	References	31

# 1. INTRODUCTION

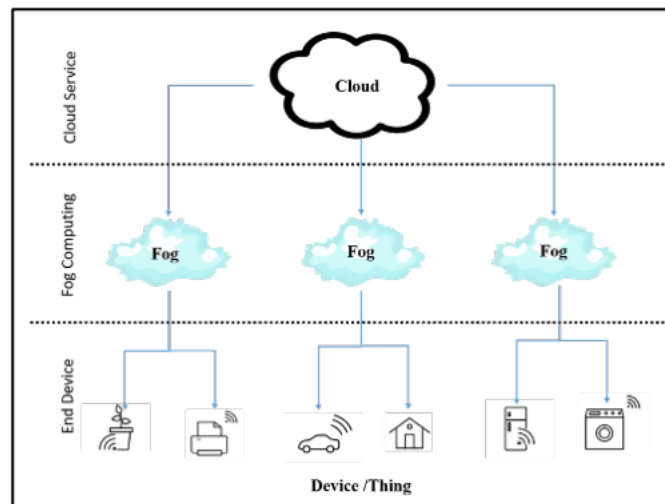
In the ever-evolving landscape of technology, the Internet of Things (IoT) stands as a paradigm that promises to reshape the way we interact with our environment and the devices that surround us. In the IoT vision, the term "things" encompasses an extensive array of objects, ranging from consumer electronic devices to household appliances, medical equipment, cameras, sensors, and more. This paradigm endeavors to seamlessly integrate these "things" into the fabric of the internet, creating a network of interconnected devices that have the potential to revolutionize our daily lives.

The IoT concept is more than just a technological trend; it is a gateway to boundless innovation. It opens the doors to new possibilities, fostering novel forms of interaction not only between these connected devices but also between these devices and humans. This interconnected world, where devices communicate intelligently with each other and with us, is the cornerstone upon which the foundations of smart cities, advanced infrastructures, and enriched services are being built. By doing so, the IoT promises to enhance our quality of life and optimize resource utilization, ultimately making a profound impact on how we live and interact with the world around us.

While the technologies and solutions facilitating connectivity and data transmission within the IoT ecosystem have been experiencing rapid growth, it is crucial to recognize that not enough attention has been dedicated to one of the core objectives of the IoT paradigm: real-time analytics and decision-making. Cloud computing has gained more popularity as there is more exchange of data and information. The Internet of Things (IoT), a major tech trend, causes connecting devices to become more ubiquitous in business, government, and personal spheres. The Internet of Things (IoT) is an accolade by cloud computing with high quality caching and high definition capabilities that enable everything to be brought online. More data is being produced by bringing in all things online. This data will be processed on the cloud. This tendency can be attributed to the design of existing data analytics approaches, which are primarily tailored to manage large volumes of data effectively. However, these approaches often fall short in terms of real-time data processing and dispatching, which is a key component of achieving the full

potential of the IoT. Cloud data centres are often located distant from IoT devices. This placement results in a high communication delay.

The dynamic nature of IoT environments and its associated real-time requirements and increasing processing capacity of edge devices has lead to the evolution of the Fog computing paradigm. Fog computing extends cloud services to the edge of networks, which results in latency reduction through geographical distribution of IoT application components, and increased scalability for handling large-scale deployments.



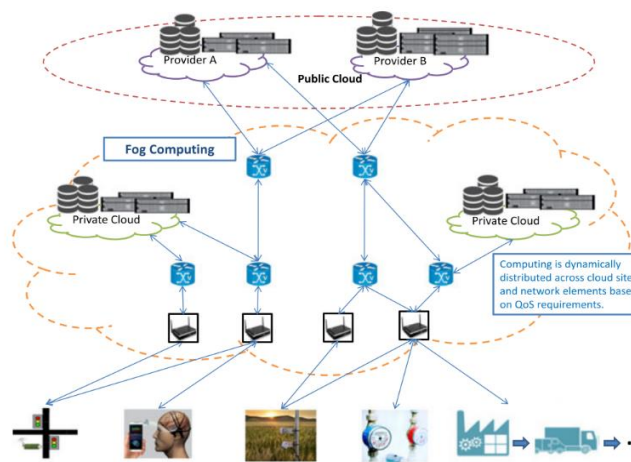
**Figure 1.1** Cloud Service, IoT, Fog Environment

Figure 1.1 illustrates the simple expression of the architecture of IoT, Cloud computing, and Fog computing, which consists of three layers. Firstly, the top layer cloud service is responsible for processing all central data and contains a top tier based on the location servers. Secondly, in the middle layer, the Fog layer, many Fog nodes, and servers are located close to the connected devices. Lastly, the lower layer or edge layer uses Internet applications and things as well as advanced devices. This layer separates the connected devices to access information from the haze server instead of connecting through the central cloud server.

## 1.1 FOG COMPUTING- DEFINITION AND CONCEPT

Fog computing emerges as a transformative paradigm, a term coined by Cisco and characterized by its ability to extend the capabilities of the cloud to the very edges of networks. At its core, Fog computing seamlessly converges the infrastructure from the expanse of the cloud data center to the periphery of network-connected devices. It encompasses intermediate entities like ISP gateways, cellular base stations, and private cloud deployments, weaving them into a continuum of resources capable of servicing multiple tenants for hosting diverse applications. Fog computing empowers the geographically distributed execution of application components, employing the resources furnished by this expansive infrastructure. This distributed resource management orchestrates and administers the services provided by each device, creating a dynamic, responsive network of computing power.

Fog computing redefines the landscape of data processing and application hosting. Unlike the traditional approach of funneling all data to centralized cloud data centers, Fog computing introduces an innovative way of data management. One of its key benefits is the significant reduction in traffic transmitted over the backbone network. The uncontrolled surge in network traffic can lead to congestion and heightened latency. By leveraging the resources of edge devices, Fog computing facilitates the filtering and analysis of sensor-generated data at its source. This localized data processing approach dramatically diminishes the volume of data sent to the cloud, enhancing network efficiency.



**Figure 1.1.1** Distributed data processing in Fog Computing

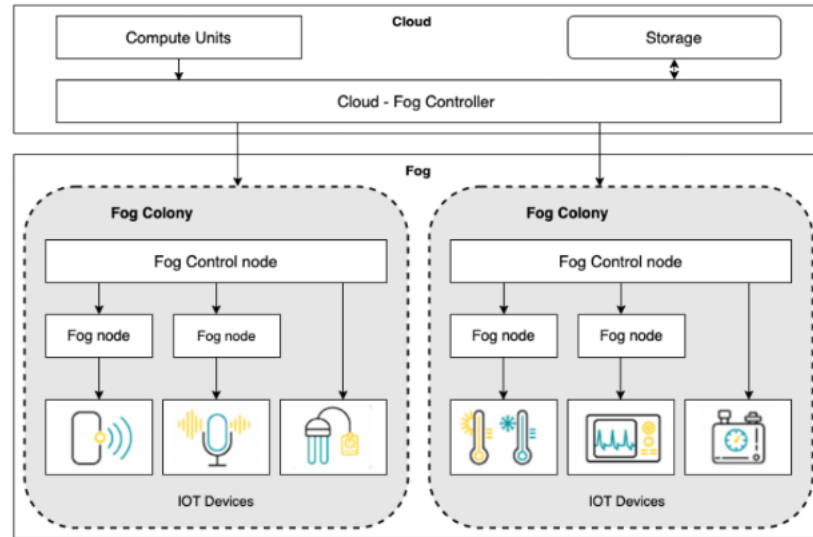
Reducing propagation latency stands as another pivotal advantage offered by the Fog computing paradigm, particularly in the context of mission-critical applications that require real-time data processing. Examples include cloud robotics, fly-by-wire aircraft control, and

vehicle anti-lock braking systems. In these applications, the timeliness of data collection, processing, and feedback is paramount. The cloud-centric model may introduce delays or even failures due to communication hiccups. Fog computing steps in by processing control systems in close proximity to devices, ensuring real-time responses and performance.

## **1.2 FOG COMPUTING ARCHITECTURE**

Fog computing, strategically positioned between the end node and the remote cloud data center, operates within a well-structured architectural framework. This architecture is notably divided into three key layers: the cloud, the fog, and the IoT sensor layer. The heart of the IoT ecosystem, IoT sensors, are responsible for capturing and transmitting data from the environment. However, they lack the computational and storage capabilities needed for processing and analysis. In conjunction with these sensors, we have actuators that enable the control of systems, responding to environmental changes sensed by the IoT sensors.

Sitting at the intermediary layer, Fog nodes play a pivotal role. These devices possess modest computational capabilities and feature network connection components like smart gateways and terminal devices. Their primary function is to collect data from IoT sensors and perform initial data reprocessing before relaying it to the upper layers. These Fog nodes operate collaboratively, sharing network, computational, and storage resources among themselves as per the specific demands of the tasks at hand. Interestingly, Fog computing is well-suited for applications requiring low-latency responses. It's important to note that the Fog devices also include cloud resources that can be provisioned on-demand from geographically distributed data sources, enhancing the ecosystem's flexibility and adaptability.



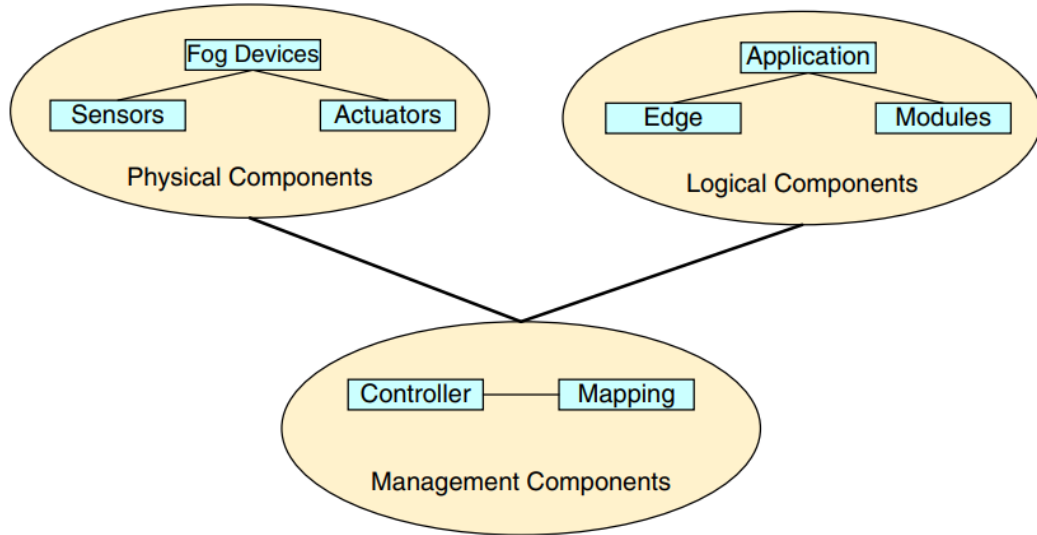
**Figure 1.2.1** Fog Architecture (Referred from “Energy efficient service placement in fog computing”)

In an extension of the basic Fog computing structure, resource and service placement gains a broader perspective. This enhanced architecture introduces two levels of control: the cloud-fog controller and the fog orchestration controller. The cloud-fog controller manages and governs all Fog cells, ensuring seamless coordination. Additionally, the fog orchestration controller plays a special role, particularly in running IoT applications, without direct reliance on the remote cloud. It takes charge of all the nodes linked to it, forming what we refer to as a "fog colony." This hierarchical structure encompasses the cloud-fog controller at the top, followed by the fog orchestration controller, the various Fog cells, and, finally, the sensor/IoT devices at the foundational bottom layer.

The controller nodes need to be provided with the information to analyze the IoT application and place the respective modules onto virtualized resources. For example, the fog orchestration controller is provided with the full information about its fog colony and the state of neighbourhood colonies. With this information, scheduler develops a service placement plan and accordingly place the application modules on particular fog resources.

### 1.3 FOG COMPUTING FRAMEWORK

The foundational CloudSim simulation toolset serves as the foundation for iFogSim [9]. One of the most used simulators for simulating cloud computing settings is called CloudSim . IoT devices (such as sensors and actuators) and customized fog computing environments can be simulated using iFogSim, which extends the abstraction of basic CloudSim classes. But with iFogSim, the classes are annotated so that users can quickly create the infrastructure, service placement, and resource allocation policies for fog computing even if they are unfamiliar with CloudSim. The distributed dataflow architecture and the Sense-Process-Actuate paradigm are applied by iFogSim to simulate any application scenario in a fog computing environment. End-to-end latency, network congestion, power consumption, operating costs, and QoS satisfaction evaluation are made easier. The management of the fog computing environment's resources , mobility , latency , quality of experience (QoE) , energy , security , and QoS-aware have previously been simulated using iFogSim in a considerable number of research papers. iFogSim is made up of three fundamental parts.



**Figure 1.3.1** High-level view of interactions among iFogSim components

#### 1.3.1. Physical Components

Fog devices, also known as fog nodes, are physical components. Hierarchical order is maintained in the fog device orchestration. The related sensors and actuators are directly coupled to the lower-level fog devices. Fog devices provide memory, network, and compute



resources, acting as the datacenters in a cloud computing model. Every fog device is built with a unique instruction processing rate and power consumption (busy and idle power) attribute that represents its capabilities and energy efficiency.

The sensors in iFogSim produce tuples, or jobs in cloud computing parlance. The interval between producing two tuples is established using deterministic distribution while building the sensors. Tuple creation is event-driven.

### **1.3.2. Logical Components**

The logical parts of iFogSim are Application edges (AppEdges) and Application modules (AppModules). Applications are viewed in iFogSim as a group of interconnected AppModules, which supports the idea of distributed applications. The features of AppEdges define the relationship between two modules. In the world of cloud computing, virtual machines (VMs) can be used to map AppModules, and AppEdges are the logical dataflow between two VMs. Each AppModule (VM) in iFogSim deals with a certain kind of tuple (task) that originates from the dataflow's previous AppModule (VM). Upon receiving a tuple of a specific type, a module can decide whether to trigger the forwarding of another tuple (of a different type) to the following module on a periodic basis.

### **1.3.3. Management Components**

Controller and Module Mapping items make up the management part of iFogSim. The Module Mapping object locates the resources in the fog devices that are available and places them inside it in accordance with the demands of the AppModules. By design, iFogSim supports module placement in hierarchies. A module is dispatched to an upper level fog device if a fog device is unable to fulfill its requirements. Following the placement instructions given by the Module Mapping object, the controller object launches the AppModules on their allocated fog devices and thereafter manages the resources of fog devices on a periodic basis. When the simulation is over, the Controller object gathers data from the fog devices about how much it cost, how much it used the network, and how much energy it used.

## **1.4 Placement Policies Problem Illustration**

In traditional cloud computing, a major problem arises due to the separation of data generation and processing, causing significant latency issues. Centralized processing leads to bandwidth congestion and compromises network efficiency. Privacy concerns surface as sensitive data is transmitted to remote cloud centers, risking breaches. Additionally, scaling centralized cloud infrastructure to accommodate growing IoT data volumes becomes costly and unsustainable.

Table below gives the problem illustration of Latency Sum. The Latency is calculated as:

**Latency** : Device to Node time+ Processing Time of node + Node to Device time

Strategy	Distribution	Placement	Latency Sum
1	(1d <sub>1</sub> , 3d <sub>2</sub> )	A: {d <sub>1</sub> ,d <sub>2</sub> }, B: {d <sub>2</sub> }, C: {d <sub>2</sub> }	21
2	(1d <sub>1</sub> , 3d <sub>2</sub> )	A: {d <sub>2</sub> }, B: {d <sub>1</sub> ,d <sub>2</sub> }, C: {d <sub>2</sub> }	26
3	(2d <sub>1</sub> , 2d <sub>2</sub> )	A: {d <sub>1</sub> }, B: {d <sub>1</sub> ,d <sub>2</sub> }, C: {d <sub>2</sub> }	23
4	(2d <sub>1</sub> , 2d <sub>2</sub> )	A: {d <sub>2</sub> }, B: {d <sub>1</sub> ,d <sub>2</sub> }, C: {d <sub>1</sub> }	25

**Table 1.4.1** Latency Sum Problem Illustration

Fog computing addresses these issues with innovative placement policies. In the above table d<sub>1</sub>,d<sub>2</sub> are the data requests and A, B,C are the placement nodes. As we can see that Latency Sum varies with Placement of data requests on different nodes. So these policies reduce latency and facilitate real-time data processing by moving it closer to the source, enhancing performance for latency-sensitive applications. They also improve bandwidth efficiency by preprocessing data at the edge, reducing congestion. By keeping sensitive data localized, Fog computing enhances security and privacy. Its distributed architecture ensures efficient resource allocation, promoting scalability without the need for extensive central cloud investments. Overall, Fog computing offers solutions that bridge the cloud-edge gap, enabling real-time capabilities while maintaining data integrity and efficiency.

## **2. LITERATURE**

Extensive research related to fog computing has been performed and we discuss the research focused on the feasible problems related to fog computing. In this section, we present the existing resource management techniques with their advantages of and limitations. A quick overview of some of these proposed job/module placement approaches is provided below.

### **2.1 Particle Swarm Optimization (PSO) in Fog Computing:**

Particle Swarm Optimization (PSO) is an optimization algorithm inspired by the collective behavior of social organisms, particularly birds flocking and fish schooling. In the context of Fog Computing, PSO is applied to tasks such as resource allocation, load balancing, and task scheduling. It involves maintaining a population of particles, where each particle represents a potential solution to the optimization problem. These particles move through the solution space, adjusting their positions and velocities based on their own best-known solution and the global best-known solution in the swarm. PSO leverages social interaction among particles, allowing them to explore and exploit the solution space more effectively. The algorithm iteratively refines solutions until it converges to an optimal or near-optimal solution.

### **2.2 Enhanced Particle Swarm Optimization (EPSO) in Fog Computing:**

Enhanced Particle Swarm Optimization (EPSO) represents an extension or enhancement of the traditional PSO algorithm, specifically tailored to address the challenges and requirements of Fog Computing environments. While EPSO shares the basic principles of PSO, it incorporates additional features and adaptations to better suit the dynamic, distributed nature of Fog Computing. For example, EPSO may integrate machine learning techniques to predict future workloads, enabling proactive resource allocation. It could also employ dynamic parameter adjustments based on real-time network conditions. EPSO algorithms aim to offer improved convergence speed, constraint handling, and adaptability to the unique characteristics of Fog Computing, where devices and workloads can change rapidly.

### **2.3 Cloud-Only Algorithm in Fog Computing:**

A Cloud-Only algorithm in the context of Fog Computing implies a centralized approach to data processing, where all computational tasks are conducted in remote cloud servers, without leveraging the computational capabilities of edge or fog devices. In this approach, data generated by edge devices is transmitted to the cloud for processing, and the results are then sent back to the edge for utilization. This approach simplifies local device management, as the edge and fog nodes primarily serve as data transmitters. However, it introduces significant challenges, notably increased latency, network dependency, and heightened bandwidth usage. While it may be suitable for applications that can tolerate higher latency and are not sensitive to network disruptions, it might not be ideal for applications requiring real-time or low-latency responses, as Fog Computing's primary advantage lies in local processing to reduce such delays. The cloud-only approach simplifies local device management but may come at the cost of potentially increased latency and network dependence, which might be unsuitable for some Fog Computing applications.

## 2.4 Comparison of Literature

Sl.no	Author (s)	Method	Advantages	Disadvantages
1	Xi Li, Yiming Liu, Hong Ji, Heli Zhang, and Victor CM Leung, 2019	Non-Orthogonal multiple access (NOMA) approach.	Decreased Computational Delay, QoS Compliance.	Complexity, NP-hard problem, Algorithm Dependency.
2	Chao Zhu, Jin Tao, Giancarlo Pastor, Yu Xiao, Yusheng Ji, Quan Zhou, Yong Li, and Antti Yla-Jaaski, 2018	Folo Proposal , Joint Optimization problem.	Latency and quality optimization, Mobility Support.	Complex Optimization, Real-World Implementation problems.
3	Zheng Chang, Liqing Liu, Xijuan Guo,	Dynamic optimization, System cost minimization.	Latency and energy improvement, Subproblem solving.	High Complexity, Resource Constraints.

	and Quan Sheng,2020			
4	Keke Gai, Xiao Qin, and Liehuang Zhu,2020	Energy-aware Fog resources Optimization, Heuristic Algorithm.	Time efficiency, Performance improvement.	Complexity, Real world deployment problems.
5	Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner,2017	Genetic algorithm, Service Placement algorithm.	Resource utilization, Reduction in network Communication delays.	Complexity, Resource constraints.

**Table 2.4.1** Comparison of Literature

### 3. PLACEMENT POLICIES OF FOG NODES

#### 3.1 JAYA algorithm

A robust scheduling algorithm can ensure efficiency of the fog resources and minimizes the energy consumption. Many previous scheduling algorithms are concerned with improving performance while ignoring the energy consumption of the fog landscape. To tackle this disadvantage, Rao implemented the JAYA algorithm with no algorithm specific parameters. The Jaya algorithm is a heuristic optimization algorithm used for solving numerical optimization problems. It is named after the Sanskrit word "Jaya," which means "victory" or "success." The Jaya algorithm is a population-based search algorithm that simulates the process of natural selection and evolution to find the optimal solution to a problem.

The algorithm operates by maintaining a population of potential solutions (individuals). In each iteration, the algorithm evaluates the fitness of each individual based on the objective function to be optimized. The key idea behind the Jaya algorithm is to improve the entire population collectively by focusing on two main principles: Exploration and Exploitation.

1. **Exploration:** The Jaya algorithm promotes exploration by moving individuals towards better solutions. It does so by comparing the fitness of two individuals within the population and moving the less fit individual closer to the better one.

2. **Exploitation:** Exploitation, on the other hand, aims to exploit the best solution found so far. In this phase, the algorithm moves all individuals towards the best solution within the population.

By iteratively applying these principles, the Jaya algorithm aims to converge towards the optimal solution. The algorithm continues its iterations until a stopping criterion is met, typically a maximum number of iterations or a satisfactory solution is found.

The Jaya algorithm is known for its simplicity, low computational cost, and ease of implementation. It is often used for solving optimization problems in various fields, such as engineering, finance, and machine learning. While it may not always outperform more sophisticated optimization algorithms, it is a valuable tool for tackling certain types of optimization problems.

### **3.2 JAYA ALGORITHM STEPS**

The JAYA algorithm is a population-based optimization method inspired by the concept of searching for the optimal solution. At its core, it mimics the iterative process of improving a population of potential solutions to find the best solution to a problem. The algorithm operates in the following manner:

#### **Algorithm Steps**

##### **Step 1: Initial population**

The initial population refers to the set of potential solutions to the optimization problem. These solutions are randomly generated or selected to kickstart the algorithm. The initial population serves as the starting point for the iterative optimization process, where each solution will be evaluated, compared, and improved over successive iterations to find the best possible solution.

## **Step 2: Fitness Evaluation of values**

The fitness evaluation step involves assessing how well each potential solution in the population performs concerning the optimization problem's objective. This is typically done by applying the objective function to each solution. The objective function quantifies the quality of a solution, helping to distinguish better solutions from weaker ones. By calculating and comparing the fitness values of the solutions, the algorithm identifies which solutions are more promising. This fitness evaluation is a crucial aspect of the algorithm, guiding the subsequent steps where solutions are improved and refined.

## **Step 3: Identify the best and worst solutions**

By comparing the fitness values of each solution we can identify the best and worst solutions. The solution with the highest fitness (i.e., the best) represents the most promising candidate. Conversely, the solution with the lowest fitness (i.e., the worst) is the least favorable one. The best solution serves as a reference point for improvement, and the worst one guides the process of replacing it with a better solution in the subsequent steps of the algorithm. This selection of the best and worst solutions plays a fundamental role in steering the optimization process toward the desired outcome.

## **Step 4: Updation**

The focus of this step is to adjust the position of each solution to converge towards the better solution (best) and diverge from the weaker solution (worst) within the population. The extent of adjustment is determined by specific rules in the algorithm. Each of the solution in the population in the JAYA algorithm solution is updated using update formula, which promotes exploration and exploitation.

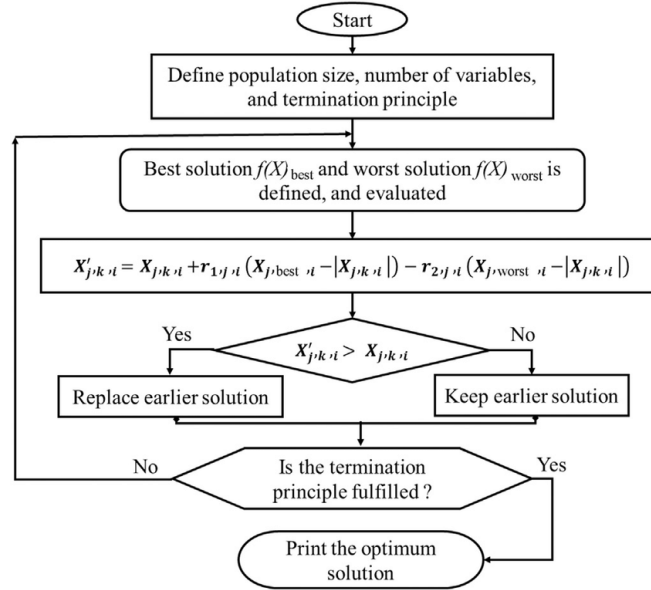
$$X_{new} = X_{j,k} + r_1(X_{best} - |X_{j,k}|) - r_2(X_{worst} - |X_{j,k}|)$$

## **Step 5: Termination criteria**

This criteria is essential for ensuring that the algorithm doesn't run indefinitely. Common termination criteria include reaching a predefined maximum number of iterations or finding a solution that meets a certain predefined quality threshold.

### Step 6: Output

This solution represents the best outcome it could achieve based on the given termination criteria. The output typically includes the values of the variables that optimize the objective function. All the particles are updated until the global optimum is found or the number of iterations is over. Finally, the solution with the highest fitness value is selected, and modules are placed on the respective fog nodes.



**Figure 3.2.1** Flowchart of JAYA Algorithm

### 3.3 Placement policy illustration of JAYA Algorithm

The below given is the problem illustration of JAYA algorithm working in different steps.

**Problem:** Minimize( $x_1 + x_2$ )

Where  $-100 \leq x_1 \leq 100$

$-100 \leq x_2 \leq 100$



**Step 1:** Initialize population

$$\begin{bmatrix} 20 & -10 \\ 58 & -28 \\ -22 & 29 \\ -1 & 2 \end{bmatrix}$$

**Figure 3.3.2** Initial Population matrix

**Step 2:** Evaluation of fitness values

Population	x <sub>1</sub>	x <sub>2</sub>	F(x)
1	20	-10	10
2	58	-28	30
3	-22	291	7
4	-1	2	1

**Table 3.3.1** Evaluation of Fitness values

**Step 3:** Identify the best and worst solution

From the Table 3.3.1 we can identify the best the worst solutions. As the objective function of the problem is minimizing ,the best solution is 1 and worst solution is 30.

Population	x <sub>1</sub>	x <sub>2</sub>	F(x)
1	20	-10	10
2	58	-28	30 (worst)
3	-22	291	7
4	-1	2	1(best)

**Table 3.3.2** Best and Worst Solutions are identified

**Step 4:** Modify and update the candidate solution

Each of the solution in the population in the JAYA algorithm solution is updated using update formula, which promotes exploration and exploitation.

$$\mathbf{X}_{\text{new}} = \mathbf{X}_{j,k} + r_1(\mathbf{X}_{\text{best}} - |\mathbf{X}_{j,k}|) - r_2(\mathbf{X}_{\text{worst}} - |\mathbf{X}_{j,k}|)$$

58	-28
-1	2

**Table 3.3.3** Best and Worst  
values

$\mathbf{x}_1$	
$r_1$	$r_2$
0.0348	0.9307

**Table 3.3.4**  
Random values for  $x_1$

$\mathbf{x}_2$	
$r_1$	$r_2$
0.9045	0.5900

**Table 3.3.5**  
Random values for  $x_2$

The above Table 3.3.3 consists the worst values and best values obtained from Table 3.3.2. In the Table 3.3.4 and Table 3.3.5  $r_1$  and  $r_2$  are the random values generated between 0 to 1. Based on these Table values we calculated the  $\mathbf{X}_{\text{new}}$  as follows:

$$\mathbf{X}_{11} = \mathbf{X}_{1,1} + r_1(\mathbf{X}_{\text{best}} - |\mathbf{X}_{1,1}|) - r_2(\mathbf{X}_{\text{worst}} - |\mathbf{X}_{1,1}|)$$

$$\begin{aligned} \mathbf{X}_{11} &= 20 + (0.0348(-1 - |20|)) - (0.9307(58 - |20|)) \\ &= -16.0974 \end{aligned}$$

$$\begin{aligned} \mathbf{X}_{12} &= -10 + (0.9045(2 - |-10|)) - (0.5900(-28 - |-10|)) \\ &= 5.184 \end{aligned}$$

$$\begin{aligned} \mathbf{X}_{21} &= 58 + (0.0348(-1 - |58|)) - (0.9307(58 - |58|)) \\ &= 55.9468 \end{aligned}$$

$$\begin{aligned} \mathbf{X}_{22} &= -28 + (0.9045(2 - |-28|)) - (0.5900(-28 - |-28|)) \\ &= -18.477 \end{aligned}$$

$$\begin{aligned} \mathbf{X}_{31} &= -22 + (0.0348(-1 - |-22|)) - (0.9307(58 - |-22|)) \\ &= -56.3056 \end{aligned}$$

$$X_{32}=29+(0.9045(2-|29|))-(0.5900(-28-|29|))$$

$$= 38.2085$$

$$X_{41}=-1+(0.0348(-1-|-1|))-(0.9307(58-|-1|))$$

$$= -54.1195$$

$$X_{42}=2+(0.9045(2-|2|))-(0.5900(-28-|2|))$$

$$= 19.7$$

Based on the above values we generated a new Table 3.3.6.

Population	$x_1$	$x_2$	$F(x)$
1	-16.0974	5.184	-10.9134
2	55.9468	-18.477	37.4698
3	-56.3096	38.2085	-18.1011
4	-54.1195	19.7	-34.4195

**Table 3.3.6**  $X_{\text{new}}$  Values

Now we compare the values of Table 3.3.1 and Table 3.3.6 to identify the best and worst solutions. The obtained values are given in Table 3.3.7. The best value for the given objective function is -34.4195 as it is the minimum value among all the  $F(x)$  values and the worst value is 30 as it is the maximum value.

Population	$x_1$	$x_2$	$F(x)$
1	-16.0974	5.184	-10.9134
2	58	-28	30

3	-56.3096	38.2085	-18.1011
4	-54.1195	19.7	-34.4195

**Table 3.3.7** Updated values of F(x)

#### **Step 5: Termination Criteria**

The steps from 1 to 4 continues until it reaches the termination criteria. Upon reaching the maximum number of iterations or on finding the optimum solution the iteration process terminates.

#### **Step 6: Output**

The output typically includes the values of the variables that optimize the objective function. The modules are placed on fog nodes based on the fitness values.

In the same way the Fog devices calculate which type of module placement scenario gives the optimized value for the given objective function and accordingly places the modules on the Fog nodes.

## **4. IMPLEMENTATION**

Program file is **ModulePlacementOnlyCloud.java**

### **4.1 Functionalities: -**

- 1) **Module Placement:** The main functionality of this code is to perform module placement in a fog computing environment.
- 2) **Calculate Module Instance Counts:** The code calculates the number of instances of each module required to be placed in the cloud.
- 3) **PSO-Based Module Placement:** The code utilizes a Particle Swarm Optimization (PSO) algorithm to optimize module placement on fog devices.
- 4) **Energy Consumption Calculation:** It calculates the energy consumption for a given module placement on fog devices.

- 5) **Global and Local Bests:** The PSO algorithm tracks both the global best solution (the best module placement) and the local best solutions for each particle.
- 6) **Dynamic Module Mapping:** The **mapModules** method dynamically maps modules to fog devices based on the PSO results, allowing for flexible and adaptive resource allocation.
- 7) **Utility and Power Analysis:** The code calculates the utilization and power consumption of fog devices, considering module placements, to make informed decisions.
- 8) **Algorithmic Flexibility:** The code demonstrates flexibility in choosing between PSO, random placement, or other algorithms for module placement, enabling experimentation and comparison.
- 9) **Update of Module Placements:** The **setNewpop** method ensures that module placements are within the valid range of available fog devices, offering robust and realistic placements.

#### **4.2 Attributes: -**

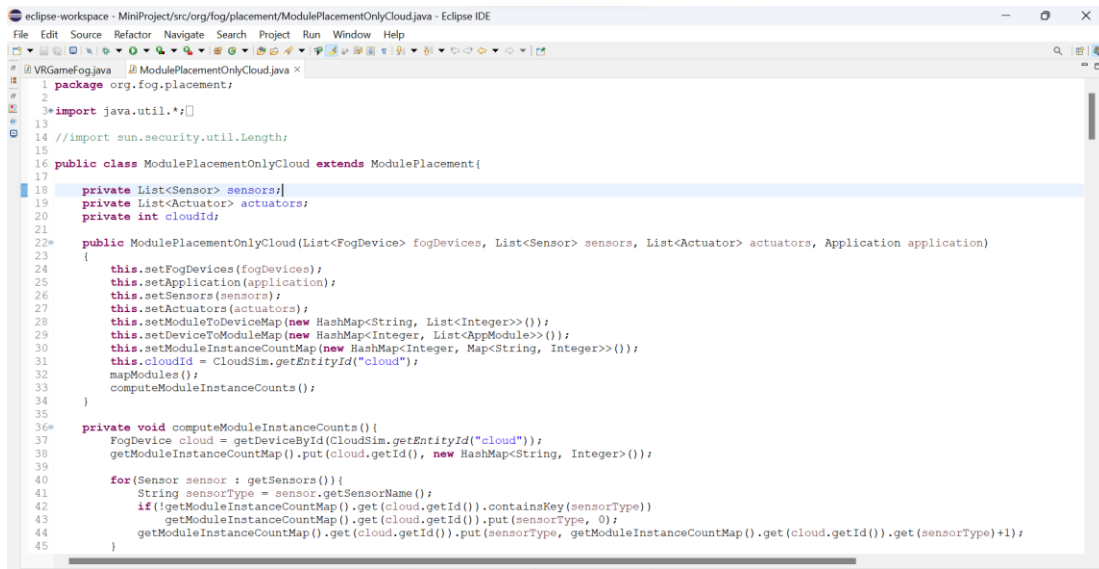
1. **sensors:** A list of Sensor objects. Sensors are devices used to collect data from the environment. This list stores the available sensors.
2. **actuators:** A list of Actuator objects. Actuators are devices used to perform actions in the environment. This list stores the available actuators.
3. **cloudId:** An integer representing the Cloud entity's ID. It stores the identifier of the cloud entity, which is used for module placement calculations involving the cloud.
4. **moduleToDeviceMap:** A mapping between module names and the fog devices where they are placed. This data structure stores information about which modules are placed on which fog devices.
5. **deviceToModuleMap:** A mapping between fog device IDs and the modules running on them. This data structure stores information about which modules are running on which fog devices.
6. **moduleInstanceCountMap:** A mapping of fog device IDs to a mapping of module names and their instance counts. It stores the count of instances for each module that is placed on a fog device.

7. **r**: An instance of the Random class for generating random numbers. It is used for various random number generation operations in the PSO algorithm.
8. **fogDevice**: A reference to a FogDevice object within loops. It represents a fog device during iterations and calculations within the code.
9. **util**: An array of Doubles representing the utilization of fog devices. It stores the utilization of each fog device's CPU, which is used in the PSO algorithm for optimizing module placement.
10. **hostEnergy**: An array of Doubles representing the energy consumption of fog device hosts. It stores the energy consumption of the host machines on fog devices, providing information for energy-aware module placement.
11. **psoorig**: An array of integers storing fog device IDs for PSO-based module placement. It represents the original set of fog devices considered for PSO-based module placement.
12. **psoorig1**: An array of integers representing a modified set of fog device IDs for PSO. It stores a modified set of fog devices used in the PSO algorithm after filtering devices with specific conditions.
13. **max**: An integer representing a maximum limit used in a loop for PSO iterations. It controls the maximum number of iterations in the PSO algorithm.
14. **n**: An integer used for tracking the best particle in the PSO algorithm. It keeps track of the best particle (module placement) found during PSO optimization.
15. **oldpop**: A 2D array of integers representing the old population of module placements in PSO. It stores the previous module placements, which are updated during each iteration of the PSO algorithm.
16. **newpop**: A 2D array of integers representing the new population of module placements in PSO. It stores the updated module placements during each iteration of the PSO algorithm.
17. **velocity**: A 2D array of integers representing the velocities of particles in the PSO algorithm. It tracks the velocities (changes) in module placements for each particle in the PSO algorithm.

18. **pBest:** An array of Doubles representing the best fitness values for each particle in PSO. It stores the best fitness values achieved by each particle in the PSO algorithm.
19. **gBest:** A Double representing the global best fitness value found in the PSO algorithm. It tracks the overall best fitness value achieved during PSO optimization.
20. **gWorst:** A Double representing the global worst fitness value found in the PSO algorithm. It tracks the overall worst fitness value achieved during PSO optimization.
21. **c1:** An integer representing the cognitive parameter used in the PSO algorithm. It controls the influence of a particle's individual best (cognitive) information on its movement in the PSO algorithm.
22. **c2:** An integer representing the social parameter used in the PSO algorithm. It controls the influence of the swarm's best (social) information on a particle's movement in the PSO algorithm.
23. **iw:** A Double representing the inertia weight used in the PSO algorithm. It controls the balance between the particle's previous velocity and the cognitive and social components in the PSO algorithm.
24. **oldFitness:** An array of Doubles storing the fitness values for the old population of module placements in PSO. It tracks the fitness values associated with the old module placements during PSO optimization.
25. **newFitness:** An array of Doubles storing the fitness values for the new population of module placements in PSO. It tracks the fitness values associated with the updated module placements during PSO optimization.
26. **map:** A LinkedHashMap mapping fog device IDs to their current utilization values. It keeps track of the utilization of each fog device during the PSO optimization process.
27. **k:** An integer used to index fog devices and modules during the PSO-based placement. It helps keep track of the current fog device or module being processed during module placement.
28. **power1:** A Double representing the power consumption of a fog device's host. It calculates the power consumption of a host machine on a fog device based on its utilization.

29. **maxIndex:** An integer used to identify the index of the maximum fitness value during PSO optimization. It helps identify the best fitness value among particles in the PSO algorithm.
30. **minIndex:** An integer used to identify the index of the minimum fitness value during PSO optimization. It helps identify the worst fitness value among particles in the PSO algorithm.

### 4.3 ENVIRONMENT SCREENSHOTS

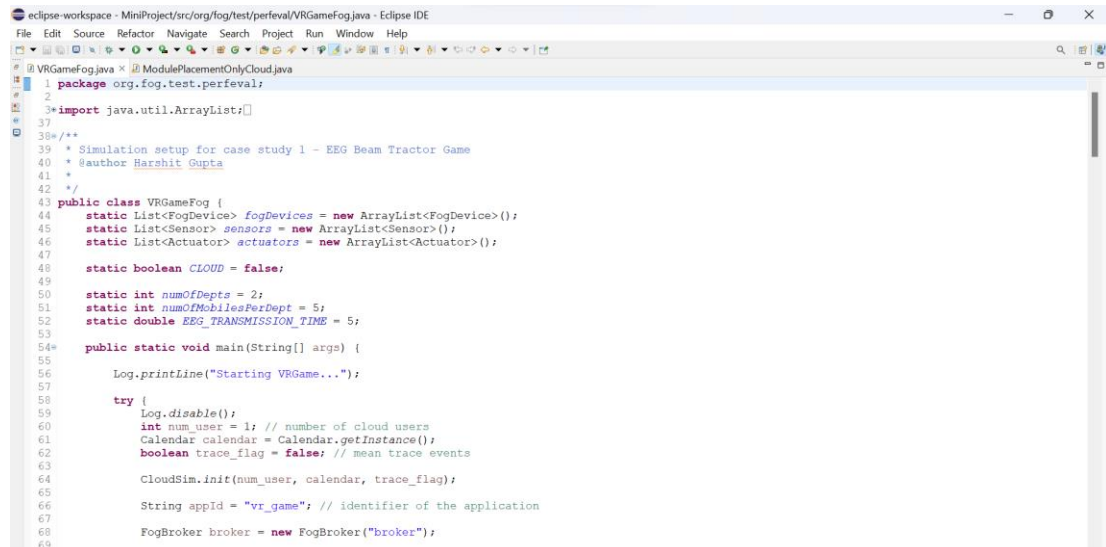


```

1 package org.fog.placement;
2
3 import java.util.*;
4
5 //import sun.security.util.Length;
6
7 public class ModulePlacementOnlyCloud extends ModulePlacement{
8
9     private List<Sensor> sensors;
10    private List<Actuator> actuators;
11    private int cloudId;
12
13    public ModulePlacementOnlyCloud(List<FogDevice> fogDevices, List<Sensor> sensors, List<Actuator> actuators, Application application)
14    {
15        this.setFogDevices(fogDevices);
16        this.setApplication(application);
17        this.setSensors(sensors);
18        this.setActuators(actuators);
19        this.setModuleToDeviceMap(new HashMap<String, List<Integer>>());
20        this.setDeviceToModuleMap(new HashMap<Integer, List<AppModule>>());
21        this.setModuleInstanceCountMap(new HashMap<Integer, Map<String, Integer>>());
22        this.cloudId = CloudSim.getEntityId("cloud");
23        mapModules();
24        computeModuleInstanceCounts();
25    }
26
27    private void computeModuleInstanceCounts() {
28        FogDevice cloud = getDeviceById(CloudSim.getEntityId("cloud"));
29        getModuleInstanceCountMap().put(cloud.getId(), new HashMap<String, Integer>());
30
31        for(Sensor sensor : getSensors()){
32            String sensorType = sensor.getSensorName();
33            if(!getModuleInstanceCountMap().get(cloud.getId()).containsKey(sensorType))
34                getModuleInstanceCountMap().get(cloud.getId()).put(sensorType, 0);
35            getModuleInstanceCountMap().get(cloud.getId()).put(sensorType, getModuleInstanceCountMap().get(cloud.getId()).get(sensorType)+1);
36        }
37    }
38
39 }

```

Figure 4.3.1 ModulePlacementOnlyCloud.java file



```

1 package org.fog.test.perfeval;
2
3 import java.util.*;
4
5 /**
6  * Simulation setup for case study 1 - EEG Beam Tractor Game
7  * @author Harshit Gupta
8  */
9
10 public class VRGameFog {
11     static List<FogDevice> fogDevices = new ArrayList<FogDevice>();
12     static List<Sensor> sensors = new ArrayList<Sensor>();
13     static List<Actuator> actuators = new ArrayList<Actuator>();
14
15     static boolean CLOUD = false;
16
17     static int numOfDepts = 2;
18     static int numOfMachinesPerDept = 5;
19     static double EEG_TRANSMISSION_TIME = 5;
20
21     public static void main(String[] args) {
22         Log.println("Starting VRGame...");
23
24         try {
25             Log.disable();
26             int num_user = 1; // number of cloud users
27             Calendar calendar = Calendar.getInstance();
28             boolean trace_flag = false; // mean trace events
29
30             CloudSim.init(num_user, calendar, trace_flag);
31
32             String appid = "vr_game"; // identifier of the application
33
34             FogBroker broker = new FogBroker("broker");
35
36         } catch (Exception e) {
37             e.printStackTrace();
38         }
39     }
40 }

```

Figure 4.3.2 VRGameFog.java file



```

===== RESULTS =====
=====
EXECUTION TIME : 819
=====
APPLICATION LOOP DELAYS
=====
[EEG, client, concentration_calculator, client, DISPLAY] ---> 226.43568782320258
=====
TUPLE CPU EXECUTION DELAY
=====
PLAYER_GAME_STATE ---> 0.3232144601003516
EEG ---> 3.6861300678684223
CONCENTRATION ---> 0.146098503629733
_SENSOR ---> 0.5991628448447904
GLOBAL_GAME_STATE ---> 0.05600000000004002
=====
cloud : Energy Consumed = 3233713.3979999577
proxy-server : Energy Consumed = 166866.59999999995
d-0 : Energy Consumed = 166866.59999999995
m-0-0 : Energy Consumed = 174784.63099999877
m-0-1 : Energy Consumed = 174771.02224874997
m-0-2 : Energy Consumed = 174760.3720599992
m-0-3 : Energy Consumed = 174668.96202249944
m-0-4 : Energy Consumed = 174630.68903999997
d-1 : Energy Consumed = 166866.59999999995
m-1-0 : Energy Consumed = 174537.55221999952
m-1-1 : Energy Consumed = 174789.7209999992
m-1-2 : Energy Consumed = 174703.09937999968
m-1-3 : Energy Consumed = 174658.25647999992
m-1-4 : Energy Consumed = 174789.72099999932
Cost of execution in cloud = 807694.9440000197
Total network usage = 196404.5

```

**Figure 4.3.3** Output file

## 4.4. EXPERIMENTAL SETUP

Setting up an experimental environment for fog computing placement policies in the iFogSim framework using the Eclipse IDE involves several steps.

### 1. Install Eclipse IDE:

- If you haven't already, download and install the Eclipse IDE. Make sure it's compatible with Java development.

### 2. Install Java Development Kit (JDK):

- Ensure you have the Java Development Kit (JDK) installed on your system. Eclipse will need this to run Java-based projects.

### 3. Download iFogSim Framework:

- Download the iFogSim framework and its source code from the official repository or website.

#### **4. Create an Eclipse Workspace:**

- Open Eclipse and create a new workspace or select an existing one to work in.

#### **5. Create a New Java Project:**

- In Eclipse, create a new Java project for your fog computing placement policies. Name the project appropriately.

#### **6. Import iFogSim Library:**

- Import the iFogSim library into your project. You can do this by right-clicking on your project in the Eclipse IDE, selecting "Build Path," and then "Add External Archives." Locate the iFogSim library JAR files and add them to your project.

#### **7. Implement Placement Logic:**

- Implement the placement logic within each placement policy in the **ModulePlacementOnlyCloud.java** file. This logic defines how modules should be placed on fog devices. You'll need to override methods or add your custom logic to control module placement.

#### **8. Configure Experiment Parameters:**

- Create a configuration file or code to set experiment parameters. These parameters might include the number of fog devices, sensors, actuators, and application details. Set up experiment-specific parameters for evaluating the placement policies.

#### **9. Run Experiments:**

- Create experiment scenarios and run experiments using your placement policies. You can create different scenarios to evaluate the policies under various conditions.

#### **10. Collect and Analyze Results:**

- Collect and analyze the results generated by your experiments. Evaluate the performance of your placement policies based on metrics like energy consumption, response time, or resource utilization.

This experimental setup will allow you to test and evaluate different fog computing placement policies using the iFogSim framework in the Eclipse IDE.

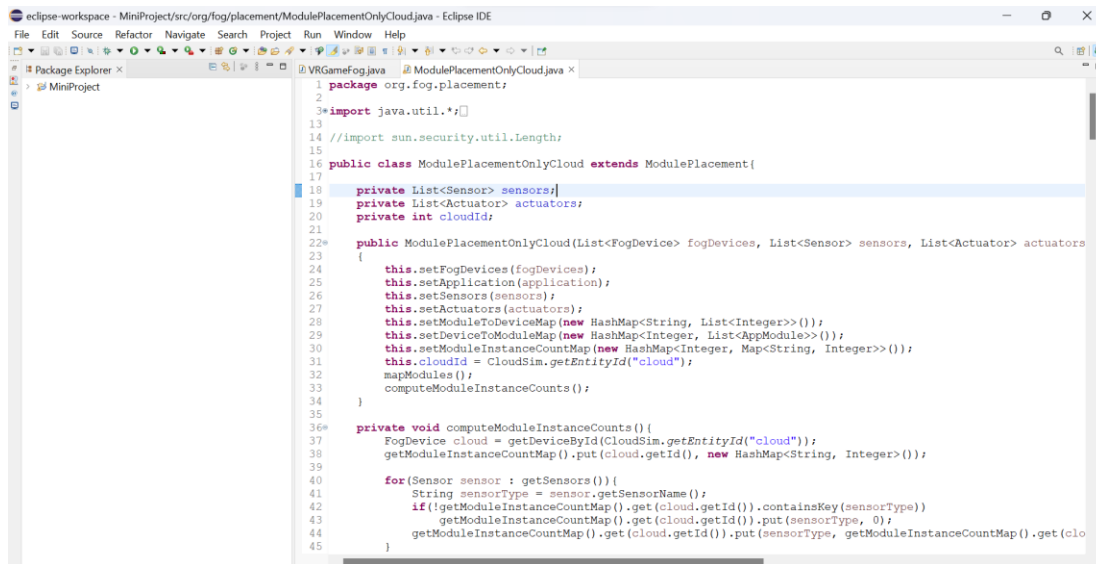


Figure. 4.4.1 Eclipse IDLE

## 4.5 PARAMETERS

Placement policies in fog computing involve, two critical parameters that are commonly considered for evaluation are **latency** and **energy consumption**. These parameters play a crucial role in assessing the effectiveness of different placement policies.

### 4.5.1 Latency

**Latency** in fog computing refers to the delay or time interval between data generation at a source and its receipt at a destination within the fog infrastructure. It is a critical parameter as it directly affects the responsiveness and performance of time-sensitive applications. Low latency is essential in applications like real-time sensor data processing, autonomous vehicles, and telemedicine, ensuring quick decision-making and a seamless user experience. High latency can lead to delays and inefficiencies, negatively impacting the effectiveness of fog

computing. Latency is influenced by network latency, processing time, communication efficiency, and more.

The latency (L) can be evaluated using the formula:

**Latency** : Device to Node time+ Processing Time of node + Node to Device time

#### 4.5.2 Energy Consumption

**Energy Consumption** is a measure of the power or energy used by fog devices within a computing system. It is a critical parameter due to its impact on both the sustainability and cost-efficiency of fog computing systems. Lower energy consumption is essential for creating energy-efficient and environmentally sustainable systems, reducing operational costs, and prolonging the operational lifespan of fog devices. In battery-powered devices, optimizing energy consumption can extend the time between recharges and reduce the frequency of maintenance. Factors influencing energy consumption include the way modules are placed on fog devices, the extent to which a device's resources are utilized, and the efficiency of hardware components. The formula for energy consumption (E) can be represented as:

**Energy Consumption (E) = Power (P) x Time (t)**

Here, Power (P) is the rate of energy usage (typically in watts), and Time (t) is the duration over which the power is consumed (typically in hours). Reducing power usage (P) and the duration of operation (t) can lead to energy-efficient fog computing systems.

## 5. DISCUSSION OF RESULT

### 5.1 Latency

The Table 5.1.1 below gives us the information about the execution times taken by each algorithm. We can clearly understand that "Cloud-Only" policy has the longest execution time (1128 ms), indicating heavy reliance on cloud-based processing. It's suitable for non-real-time applications but lacks low-latency performance. "PSO" policy significantly reduces execution time to 830 ms, optimizing module placement and reducing cloud dependency. "EPSO" further enhances performance with an execution time of 819 ms, suggesting additional optimization techniques." JAYA" policy stands out with the shortest execution time (805 ms) and efficient module allocation, making it ideal for low-latency, real-time applications with strict timing requirements.

Method	Execution Time (ms)
CloudOnly	1128
PSO	830
EPSO	819
JAYA	805

**Table 5.1.1** Latencies of Different algorithms

The chart below presents the results of execution for both the existing placement policies like EPSO, PSO, Cloud-Only, and the new JAYA algorithm. In this analysis, it's evident that the execution times of the existing methods are consistently higher compared to the JAYA algorithm. Upon closer examination, it becomes clear that the JAYA algorithm consistently generates lower execution times, resulting in significantly lower latency. This trend indicates the enhanced productivity and effectiveness of the JAYA algorithm compared to the existing placement policies like EPSO, PSO, and Cloud-Only.



**Figure 5.1.1** Execution Times of different Algorithm

## 5.2 Energy Consumption

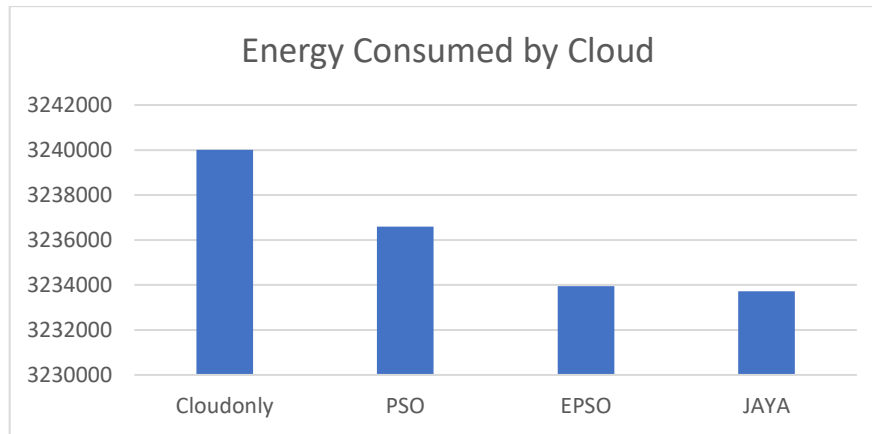
The data provided in the Table 5.2.1 represents the energy consumed by different placement policies in a fog computing system, and the values are measured in a unit like watts per unit of time (e.g., watt-seconds). The "Cloud-Only" policy consumes the highest amount of energy, totaling approximately 3,240,001.56 watt-seconds. This is expected, as this policy primarily relies on cloud resources, which tend to consume more power due to data transmission and processing.

Among the optimization-based policies, "PSO" consumes slightly less energy, with an energy consumption of around 3,236,598.59 watt-seconds. "EPSO" and "JAYA" exhibit similar energy consumption levels, with values of about 3,233,950.58 and 3,233,713.40 watt-seconds, respectively. These results suggest that these optimization-based policies aim to allocate tasks more efficiently to fog devices, reducing overall energy consumption compared to the "Cloud-Only" approach.

Method	Energy consumed by cloud
Clouonly	3240001.555357
PSO	3236598.59085709
EPSO	3233950.57857138
JAYA	3233713.39799957

**Table 5.2.1** Energy consumed by cloud

The chart below shows the energy consumed by cloud. It gives the detailed information, "Cloud-Only" exhibits the highest energy consumption, indicated by the tallest bar. This aligns with the expectation that a policy relying heavily on cloud resources consumes more energy due to data transmission and processing in the cloud. "PSO" policy demonstrates a noticeable reduction in energy consumption compared to "Cloud-Only." It is the second tallest bar, indicating more efficient energy utilization through optimization. "EPSO" and "JAYA" show very similar energy consumption levels, both notably lower than "Cloud-Only." These optimization-based policies aim to allocate tasks more efficiently to fog devices, reducing overall energy consumption.



**Figure 5.2.1** Energy Consumed by cloud in Different Algorithms

## 6. CONCLUSION

In conclusion, the project's exploration of placement policies in fog computing, with a specific focus on the innovative JAYA algorithm, has yielded significant insights and promising outcomes. The JAYA algorithm has proven to be a game-changer in optimizing module allocation within the fog computing infrastructure, resulting in notable benefits for the system's performance and efficiency. It efficiently allocates modules to fog devices, reducing reliance on cloud resources and leading to lower latency, a crucial aspect for real-time applications with stringent timing requirements. Moreover, the JAYA algorithm has demonstrated impressive energy efficiency, contributing to cost-effectiveness and sustainability by reducing energy consumption. This makes it a promising approach for building eco-friendly fog computing systems.

Comparatively, the JAYA algorithm outperforms traditional placement policies like "Cloud-Only," "PSO," and "EPSO" by consistently delivering lower execution times and reduced energy consumption. Its comparative advantage positions JAYA as the preferred choice for applications demanding low latency and optimized resource utilization. The success of the JAYA algorithm in this project opens doors to further exploration and integration into real-world fog computing systems. It offers an exciting pathway to harness the full potential of fog computing by achieving low latency, optimizing energy consumption, and enhancing overall system performance. This holds significant promise for the future of fog computing and its application in time-sensitive domains, including IoT, edge computing, and autonomous systems.

## 7. REFERENCE

- [1] Xi Li, Yiming Liu, Hong Ji, Heli Zhang, and Victor CM Leung. Optimizing resources allocation for fog computing-based internet of things networks. *IEEE Access*, 7:64907–64922, 2019.
- [2] Chao Zhu, Jin Tao, Giancarlo Pastor, Yu Xiao, Yusheng Ji, Quan Zhou, Yong Li, and Antti Ylä-Jääski. Folo: Latency and quality optimized task allocation in vehicular fog computing. *IEEE Internet of Things Journal*, 6(3):4150–4161, 2018.
- [3] Zheng Chang, Liqing Liu, Xijuan Guo, and Quan Sheng. Dynamic resource allocation and computation offloading for iot fog computing system. *IEEE Transactions on Industrial Informatics*, 17(5):3348–3357, 2020.
- [4] Keke Gai, Xiao Qin, and Liehuang Zhu. An energy-aware high performance task allocation strategy in heterogeneous fog computing environments. *IEEE Transactions on Computers*, 70(4):626–639, 2020.
- [5] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized iot service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, 2017.
- [6] V Dinesh Reddy, Kurinchi Nilavan, GR Gangadharan, and Ugo Fiore. Forecasting energy consumption using deep echo state networks optimized with genetic algorithm. In *Artificial Intelligence, Machine Learning, and Data Science Technologies*, pages 205–217. CRC Press, 2021.
- [7] R Rao. Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. *International Journal of Industrial Engineering Computations*, 7(1):19–34, 2016.



- [8] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.
- [9] Narayana Potu, Chandrashekar Jatoth, and Premchand Parvataneni. Optimizing resource scheduling based on extended particle swarm optimization in fog computing environments. *Concurrency and Computation: Practice and Experience*, page e6163, 2021.
- [10] Xiaoge Huang, Weiwei Fan, Qianbin Chen, and Jie Zhang. Energy-efficient resource allocation in fog computing networks with the candidate mechanism. *IEEE Internet of Things Journal*, 7(9):8502–8512, 2020.