InvestiGO
Revista Científica Multidisciplinaria
ISSN 2953-6367
http://revistainvestigo.com

Julio - diciembre 2026
Vol. 7 No. 20, PP. 380-403
https://doi.org/10.56519/mh51aq58

SIMULACIÓN DE LA TOPOLOGÍA DE RED EN ENTORNOS IFOG

NETWORK TOPOLOGY SIMULATION IN IFOG ENVIRONMENTS

Jaime David Camacho Castillo[1], Ángel Patricio Flores Orozco[2], Jonny Israel Guaiña Yungán[3], Diana Elizabeth Gómez García[4]

{jaimed.camacho@espoch.edu.ec[1], aflores@espoch.edu.ec[2], jguaina@espoch.edu.ec[3], elizabeth.gomez@espoch.edu.ec[4]}

Fecha de recepción: 12/05/2026 / Fecha de aceptación: 22/06/2026 / Fecha de publicación: 08/07/2026

RESUMEN: Este trabajo aborda el desafío crítico de la colocación dinámica de microservicios en un entorno Fog heterogéneo, un problema fundamental para garantizar la eficiencia en aplicaciones modernas de Internet de las Cosas (IoT). Debido a la naturaleza de las limitaciones del cloud computing tradicional, como la latencia variable y el alto consumo de energía, se propone un método basado en Particle Swarm Optimization (PSO). La propuesta combina un diseño de tres capas (IoT, Fog, Cloud) basado en el modelo de rendimiento generado por el simulador iFogSim2, lo que facilita la evaluación de la distribución de la carga en nodos con diferentes habilidades. Los resultados principalmente indican que la política basada en PSO obtiene mejoras considerables en la latencia de extremo a extremo, reduciéndola entre un 20% y 39% contra estrategias estáticas tales como Round-Robin y Random. Además, el consumo de energía se reduce de forma consistente, con una mejora de hasta un 25% bajo alta carga. Los resultados del análisis estadístico ANOVA y t de Welch verifican que estas mejoras son significativas con valores p menores a 10^{-12}, corroborando la robustez de la metaheurística adaptativa. Se demuestra que el PSO no sólo mejora el rendimiento promedio, sino también asegura la estabilidad del sistema, y disminuye dramáticamente las violaciones SLA; siendo esto indicador de que es la mejor opción para la gestión de recursos en infraestructuras de borde heterogéneas.

Palabras clave: Fog computing, microservicios, particle swarm optimization, iFogSim2, eficiencia energética, latencia

ABSTRACT: This work addresses the critical challenge of dynamic microservice placement in a heterogeneous Fog environment, a fundamental problem for ensuring efficiency in modern Internet of Things (IoT) applications. Due to the nature of traditional cloud computing limitations, such as variable latency and high energy consumption, a method based on Particle Swarm Optimization (PSO) is proposed. The proposal combines a three-layer design (IoT, Fog, Cloud) based on the performance model generated by the iFogSim2 simulator, facilitating the

[1]Escuela Superior Politécnica de Chimborazo ESPOCH, Riobamba – Ecuador, https://orcid.org/0000-0002-9110-6585
[2]Escuela Superior Politécnica de Chimborazo ESPOCH, Riobamba – Ecuador, https://orcid.org/0000-0003-1484-2949
[3]Escuela Superior Politécnica de Chimborazo ESPOCH, Riobamba – Ecuador, https://orcid.org/0000-0003-0456-7429
[4]Escuela Superior Politécnica de Chimborazo ESPOCH, Riobamba – Ecuador, https://orcid.org/0009-0004-1182-6980

evaluation of load distribution across nodes with different capabilities. The results mainly indicate that the PSO-based policy achieves significant improvements in end-to-end latency, reducing it between 20% and 39% compared to static strategies such as Round-Robin and Random. Furthermore, energy consumption is consistently reduced, with an improvement of up to 25% under high load conditions. The results of the ANOVA and Welch's t-test statistical analyses verify that these improvements are significant, with p-values lower than 10^{-12}, corroborating the robustness of the adaptive metaheuristic. It is demonstrated that PSO not only improves average performance but also ensures system stability and dramatically decreases SLA violations, indicating that it is the best option for resource management in heterogeneous edge infrastructures.

Keywords: Fog computing, microservices, particle swarm optimization, iFogSim2, energy efficiency, latency

INTRODUCCIÓN

El crecimiento sin precedentes de dispositivos conectados bajo el paradigma del Internet de las Cosas (IoT), ha transformado radicalmente la demanda de servicios computacionales; ya que tradicionalmente, la computación en la nube ha sido la principal opción para procesar datos, pero en ambientes con sensores ubicuos, tráfico continuo, y necesidad de decisiones en tiempo real, este esquema centralizado sufre cuellos de botella severos. Entre estos problemas destacan latencias variables debido a la congestión de la red troncal, dependencia de enlaces de comunicación intermitentes y un incremento insostenible del consumo energético asociado a la transmisión masiva de datos hacia centros de datos remotos. En respuesta a estas limitaciones, el desplazamiento del procesamiento hacia el borde de la red, bajo los paradigmas de Edge y Fog computing, se ha consolidado como una alternativa técnica viable para reducir el retardo extremo a extremo, aliviar la red troncal y habilitar servicios sensibles al contexto (1).

El Fog Computing introduce la arquitectura de nivel medio entre los dispositivos de IoT y la nube, que posee características como la proximidad, distribución geográfica y heterogeneidad de recursos; pues el Fog fue concebido como una extensión del cloud hacia el borde con baja latencia, conciencia de ubicación y soporte para movilidad desde sus primeras propuestas (2). En un despliegue real de Fog Computing, la infraestructura no es un clúster homogéneo, sino un mosaico de gateways y microservidores con límites estrictos de CPU y energético, que convierte la ubicación de servicios en un problema dinámico de asignación (3). La transición a microservicios amplifica esta complejidad, haciendo que cumplir con QoS sea gestionado simultáneamente a que se utilicen eficientemente los recursos, donde las metaheurísticas cobran relevancia como soluciones potenciales ante problemas de esta índole (4, 5).

Dada la importancia estratégica de estas infraestructuras, se ha desarrollado una amplia gama de marcos de simulación y herramientas. Estudios exhaustivos han revisado estas direcciones de investigación, destacando la necesidad de marcos que capturen la heterogeneidad (6). Entre ellos, el kit de herramientas iFogSim ha sido básico para el modelaje y simulación de recursos en la red de niebla (7). En fechas recientes, se han planteado plataformas como Simcan2Fog, una

plataforma de eventos discretos que permite un modelado detallado de estos ambientes (8). Asimismo, teniendo en cuenta la evolución tecnológica, se desarrolló CloudSim 7G, que presenta una arquitectura renovada, con un mejoramiento significativo en el tiempo de ejecución y una mayor eficiencia en el uso de la memoria (9). Otros marcos destacados incluyen FogNetSim++, diseñado para simular ambientes Fog distribuidos (10), e IFCSim, diseñado para analizar el rendimiento de diversas topologías (11).

Las investigaciones actuales en el área de orquestación para Fog computing, se centran en eliminar las limitaciones de rígida operatividad por medio de arquitecturas basadas en microservicios y métodos adaptativos distribuidos (5). Pero, a pesar del aumento en la creación de simuladores de nueva generación y modelos combinatorios para balancear cargas de cálculo (7, 10), las propuestas actuales siguen sufriendo de un gap fatal entre tratar el balance energético y el cumplimiento de los acuerdos de nivel de servicio (SLA) de manera desagregada. Casi toda la literatura actual simplifica la fluctuación de la potencia en el dispositivo en el borde o no incluye planes de penalización estrictos en caso de saturar un nodo, dejando un vacío técnico en situaciones con tráfico estocástico y alta heterogeneidad física (11).

Uno de los retos persistentes en estas arquitecturas es enrutamiento/paket colocación de nodo. Si bien algunos trabajos se aproximan al enrutamiento realista sin protocolos tradicionales (12), otros trabajan específicamente en la ubicación de nodos Fog considerando baja latencia (13). Se han estudiado en la literatura diferentes algoritmos para la asignación de tareas (14), así como modelos híbridos que emplean metaheurísticas tales como JAYA-GA para la ubicación óptima de nodos (15). En este sentido, la optimización de las métricas de rendimiento únicamente puede dar lugar a un aumento significativo del consumo total de energía, lo que comprometería la sostenibilidad (16). Por tanto, en el presente trabajo se propone aplicar Particle Swarm Optimization (PSO), un algoritmo basado en el comportamiento social de enjambres, para la solución de la problemática de colocación de microservicios tratando de balancear el rendimiento y la eficiencia energética de manera dinámica a través de simuladores avanzados de última generación.

MATERIALES Y MÉTODOS

La presente sección detalla la metodología empleada para abordar el problema de la colocación dinámica de microservicios, estructurándose bajo los criterios de rigor científico y exhaustividad técnica requeridos en la simulación de entornos de red complejos.

Categoría de la Investigación

Este estudio se clasifica como una investigación cuantitativa de carácter experimental, basada en la simulación controlada de sistemas distribuidos, y está basado en un método deductivo, asumiendo como hipótesis que las políticas adaptativas basadas en metaheurísticas son mejores que los métodos estáticos en cuanto a latencia y consumo energético en un entorno heterogéneo. La metodología adopta un diseño factorial de experimentos en el que se manipulan variables como el nivel de carga (bajo, medio, alto) y la política de colocación para evaluar el impacto en métricas de Calidad de Servicio (QoS) y sostenibilidad.

Descripción de la Investigación

La investigación se centra en el diseño y evaluación de un algoritmo de optimización para la asignación de microservicios en una topología IoT-Fog-Cloud. El problema se modela como una optimización multiobjetivo con restricciones físicas de CPU, RAM y energía.

- **Diseño de la Arquitectura:** Se implementó una infraestructura de tres niveles. El nivel inferior (IoT) es donde los sensores producen datos a una tasa variable dependiendo del escenario de carga. El nivel medio (Fog) está formado por nodos con capacidad computacional de 4000, 3000 y 2000 MIPS. Nivel superior (Nube) proporciona un respaldo elástico con alta latencia de comunicaciones.

- **Algoritmo de Optimización:** Se escogió Particle Swarm Optimization (PSO) por su eficacia para resolver problemas de optimización combinatoria en ambientes distribuidos y elásticos (17). La arquitectura del enjambre consiste en una población fija de $N = 20$ partículas y un máximo de $I_{max} = 60$ iteraciones. Estos parámetros se establecieron formalmente a través de un análisis de sensibilidad y convergencia previa, monitoreando el desempeño del algoritmo en un espacio de $N \in (10, 50)$ e iteraciones $I \in (40, 150)$. Los análisis demostraron, de forma concluyente, que un tamaño de enjambre de 20 partículas proporciona la óptima diversidad estocástica para evitar mínimos locales y no genera una explosión combinatoria, mientras que 60 iteraciones proporcionan estabilidad asintótica del algoritmo. Este tamaño reduce el overhead computacional impreso del proceso de optimización, manteniéndolo en el sub-milisegundo, una necesidad para los esquemas de proveer tiempo real en la capa de Fuego. Cada partícula codifica vectorialmente una matriz de asignación lógica que representa el mapeo entre microservicios y los nodos físicos disponibles, siguiendo las directrices metodológicas de trabajos recientes en la técnica (18).

- **Función de Fitness:** Con el objetivo de guiar la búsqueda metaheurística hacia topologías eficientes y físicamente viables, el rendimiento de cada solución candidata x se evalúa mediante una función de aptitud multiobjetivo con penalización externa, formalizada matemáticamente mediante la siguiente ecuación:

$$\text{Fitness}(x) = \alpha \cdot (L(x) / L_{max}) + (1 - \alpha) \cdot (E(x) / E_{max}) + P(x)$$

Donde L(x) representa la latencia de extremo a extremo acumulada por los flujos de datos bajo la configuración x, E(x) constituye el consumo energético total de la infraestructura, mientras que Lmax y Emax actúan como factores de normalización basados en las peores condiciones de operación del sistema. El coeficiente de ponderación se fijó de manera asimétrica en $\alpha = 0.65$ (por consiguiente, $1 - \alpha = 0.35$). Esta distribución de pesos se justifica rigurosamente en el contexto de aplicaciones IoT críticas (tales como telemedicina o control industrial), donde el retardo temporal es la métrica de Calidad de Servicio (QoS) primaria, dado que superar los tiempos límite de ejecución (deadlines) invalida la integridad del sistema; no obstante, el peso remanente de 0.35 mantiene una presión selectiva firme hacia la sostenibilidad energética. El término P(x) define la

función de penalización estricta frente a la violación de restricciones físicas de hardware, descrita matemáticamente mediante la siguiente función por tramos:

$$P(x) = \begin{cases} 0 \text{ , si } \forall\, n \in NFog : Demand(n) \leq Capacity(n) \\ W\_OVERLOAD \text{ , en caso contrario} \end{cases}$$

Donde W_OVERLOAD = 1,000,000 es una constante de penalización severa. Este mecanismo matemático introduce un costo prohibitivo en el valor de aptitud, forzando al enjambre a descartar inmediatamente cualquier solución que sature las capacidades de CPU o memoria en los nodos Fog, asegurando la viabilidad operativa de la topología dentro del entorno simulado bajo iFogSim2.

Aplicación en el Estudio

Para la ejecución de los experimentos, se utilizó el toolkit iFogSim2 (19), una extensión avanzada de iFogSim (7) que permite el modelado de movilidad y gestión de microservicios. La aplicación práctica de estas herramientas permitió:

- **Modelado de Microservicios:** La aplicación IoT se representó como un Grafo Acíclico Dirigido (DAG), donde cada nodo es un microservicio con demandas específicas de procesamiento (MIPS).
- **Simulación de Cargas:** Se configuraron flujos de datos dinámicos mediante AppEdges de tipo SENSOR y ACTUATOR, permitiendo capturar el retardo extremo a extremo percibido por el usuario final bajo modelos de interoperabilidad avanzados (20).
- **Configuración de Energía:** La configuración de Energía implica la aplicación de modelos de potencia lineal (FogLinearPowerModel) para calcular el consumo energético dependiendo del uso (estado de la unidad de procesamiento en estado de espera u ocupada).
- **Garantía de Reproducibilidad:** La reproducibilidad implica una configuración que se establezca con un valor determinista de semilla (SEED = 42L) para los generadores de números aleatorios que permiten mantener un entorno experimental consistente, con la misma carga de tráfico y recursos disponibles tanto en la implementación de PSO como en los sistemas basados en Round-Robin y en Random.

RESULTADOS

Los resultados obtenidos a través de la simulación completa en iFogSim2 ofrecen una representación cuantitativa del comportamiento del sistema bajo diferentes políticas de gestión y condiciones de demanda. En esta sección se expone el análisis de las métricas de rendimiento, eficiencia energética y cumplimiento de niveles de servicio, adicional con una interpretación

narrativa de la información contenida en cada tabla, para apoyar las conclusiones técnicas del estudio.

Análisis Integral de Desempeño de Carga

Los resultados para cada configuración experimental se presentarán en forma de medias y desviaciones estándar de 30 ejecuciones independientes y figuran en la Tabla 1. Este método asegura que los resultados mostrados sean representativos y que las tendencias observadas sean estadísticamente robustas.

Tabla 1. Desempeño por carga y política (promedio ± desviación estándar, n=30).

Carga	Política	Latencia (ms)	Energía (J)	Throughput (tuplas/s)	Violaciones SLA	Fitness (prom ± sd)
Baja	PSO	48.95 ± 2.31	95.62 ± 5.53	716.83 ± 34.69	0.10	0.813 ± 0.032
Baja	Round-Robin	61.40 ± 4.14	111.27 ± 7.24	572.55 ± 39.96	0.33	0.997 ± 0.054
Baja	Random	69.29 ± 5.78	115.34 ± 10.35	508.73 ± 42.90	0.13	1.097 ± 0.061
Media	PSO	69.04 ± 3.72	138.81 ± 10.31	508.39 ± 27.67	0.40	0.778 ± 0.038
Media	Round-Robin	91.28 ± 7.15	168.44 ± 13.03	385.51 ± 32.77	0.07	1.001 ± 0.060
Media	Random	101.14 ± 7.84	173.23 ± 14.06	348.28 ± 26.55	0.20	1.085 ± 0.062
Alta	PSO	96.27 ± 5.03	207.99 ± 15.42	364.69 ± 19.37	0.13	0.754 ± 0.031
Alta	Round-Robin	134.55 &plus style="padding: 8px;">8.23	245.87 ± 23.69	261.10 ± 16.34	4.10	0.999 ± 0.054
Alta	Random	156.89 ± 9.33	277.92 ± 23.35	223.25 ± 13.44	11.67	1.154 ± 0.062

Como se observa en la Tabla 1, la política basada en PSO presenta un desempeño superior en todos los niveles de carga evaluados. En el escenario de **baja carga**, PSO puede alcanzar una latencia de 48.95 ms, lo que significa una reducción significativa en comparación con 61.40 ms de Round-Robin y 69.29 ms de la política Random. Esta eficiencia se refleja incluso en la energía consumida (95.62 J para PSO), y lleva a la mejor valoración de fitness (0.813), lo cual indica una colocación inicial óptima. A la carga media, la latencia de PSO incrementa a 69.04 ms, pero continua mejor que los baselines que ya se encuentran por encima de 90 ms. Finalmente, bajo **carga alta**, el sistema propuesto mantiene una latencia de 96.27 ms, mientras que Round-Robin y Random experimentan una degradación severa (134.55 ms y 156.89 ms respectivamente) y un incremento drástico en las violaciones de SLA, las cuales pasan de 0.13 en PSO a 11.67 en Random. Esto demuestra que PSO no solo optimiza el rendimiento, sino que garantiza la estabilidad del sistema bajo presión extrema.

Análisis de Mejora Relativa y Eficiencia

Para medir el beneficio competitivo del enfoque propuesto, la Tabla 2 muestra la mejora porcentual de PSO con respecto a las estrategias de referencia. Este análisis lleva a la observación del hecho de que el beneficio de la optimización inteligente se exagera cuando la carga de requisitos del entorno Fog crece.

Tabla 2. Mejora porcentual de PSO frente a baselines (promedios).

Carga	Mejora Latencia vs RR	Mejora Energía vs RR	Mejora Latencia vs Random	Mejora Energía vs Random
Baja	20.28%	14.06%	29.35%	17.10%
Media	24.37%	17.59%	31.75%	19.87%
Alta	28.45%	15.40%	38.64%	25.16%

Los datos de la Tabla 2 muestran una tendencia de crecimiento sistemático en la eficiencia de PSO, y mejoras en latencia frente a Round-Robin que sube desde 20.28% en carga baja hasta 28.45% en carga alta. Comportamiento parecido se obtiene frente a la política Random, donde la mejora es de un 38.64% ante condiciones de estrés del sistema. En cuanto a la sostenibilidad, la mejora energética respecto a Random es de un 25.16% para alta carga, lo cual es importante para despliegues de Fog reales en las que la capacidad de disipación de calor junto al costo de electricidad es factor crítico; y son estos resultados los que validan que la metaheurística adaptativa es especialmente provechosa cuando los recursos de red tienden a estar cercanos a su límite de saturación.

Validación Estadística de los Resultados

Para demostrar que las diferencias observadas entre las políticas de colocación no eran meramente fluctuaciones estocásticas de la simulación, se realizaron pruebas de hipótesis estrictas. La Tabla 3 muestra los resultados de la prueba t de Student pareada (Welch t-test) para PSO contra cada baseline para las métricas de latencia con diferentes regímenes de carga.

Tabla 3. Resultados de pruebas t de Student: PSO vs baselines por carga.

Carga	Métrica	Comparación	t	P
Baja	Latencia	PSO vs RR	-14.37	1.59e-18
Baja	Latencia	PSO vs Random	-17.89	4.00e-20
Alta	Latencia	PSO vs RR	-20.77	2.27e-24

Alta	Latencia	PSO vs Random	-29.81	8.13e-29

La interpretación de la tabla 3 es concluyente pues para todas las comparaciones de latencia p es muy pequeño ($p << 0.05$), lo que permite rechazar la hipótesis nula de igualdad de medias con un nivel muy alto de seguridad. Los valores negativos de la estadística t confirman que PSO tiene latencias consistentemente menores en comparación con las otras. Estos resultados son apoyados por el análisis global de varianza mostrado en la Tabla 4 que compara las tres políticas juntas de manera significativa.

Tabla 4. Pruebas estadísticas para contrastar PSO vs. baselines (Parte A: ANOVA de una vía).

Carga	ANOVA Latencia (F, p)	ANOVA Energía (F, p)
Baja	F=169.20, p=1.04e-30	F=51.31, p=1.91e-15
Media	F=192.73, p=1.08e-32	F=66.02, p=3.60e-18
Alta	F=469.67, p=2.39e-47	F=82.03, p=9.52e-21

Como se detalla en la Tabla 4, el estadístico F del ANOVA tiene valores muy altos, en particular para el escenario de carga alta (para latencia F=469.67), lo que es un indicativo de que la política de ubicación en desafío es un factor crítico y determinante en el rendimiento del sistema completo. Es así que la Tabla 5 completa este análisis mostrando las pruebas t de Welch más divididas enfocadas en energía y latencia para casos particulares de comparación directa.

Tabla 5. Pruebas estadísticas para contrastar PSO vs. baselines (Parte B: t-test de Welch).

Carga	Métrica	Comparación	t	P
Baja	Latencia	PSO vs Round-Robin	-14.37	1.59e-18
Alta	Energía	PSO vs Random	-12.79	1.34e-15

La Tabla 5 ratifica la consistencia del desempeño superior de PSO, confirmando que incluso en la métrica energética bajo carga alta, la diferencia frente a la política Random es estadísticamente significativa (p=1.34e-15). En conjunto, los resultados presentados en esta sección validan empíricamente que la utilización de Particle Swarm Optimization en arquitecturas Fog heterogéneas permite alcanzar un balance óptimo entre latencia y energía, superando con creces las capacidades de las estrategias de gestión convencionales.

DISCUSIÓN

El análisis exhaustivo de los resultados presentados en la sección anterior revela una superioridad técnica indiscutible de la política basada en Particle Swarm Optimization (PSO) sobre las

estrategias convencionales de Round-Robin y Random. Esta ventaja se presenta en varias dimensiones y no solo en la ejecución de computación bruta en el sistema, es decir en la latencia entre extremos, sino en la eficiencia del sistema en torno al consumo de energía y el cumplimiento de los acuerdos de nivel de servicio (SLA). Si se observa la Tabla 1, se puede apreciar cómo la transición hacia una metaheurística adaptativa posibilita que el sistema explote la heterogeneidad intrínseca de los nodos Fog, que ha sido considerada como un factor clave para el éxito de los servicios sensibles al contexto de la red.

El comportamiento diferencial observado entre las estrategias analizadas trasciende la mera

variación métrica y debe interpretarse bajo los fundamentos teóricos de la optimización adaptativa de recursos distribuidos. Mientras que las políticas estáticas convencionales inducen un colapso del rendimiento debido a su incapacidad inherente de lectura contextual, la metaheurística propuesta previene la saturación de los elementos con menor capacidad de cómputo (nodos de 2000 MIPS frente a los de 4000 MIPS) a través de su dinámico mecanismo de búsqueda socio-cognitiva. Matemáticamente, el vector de velocidad de cada partícula ajusta continuamente su trayectoria mediante la atracción ponderada hacia su mejor experiencia histórica (pbest) y el óptimo global descubierto por el enjambre (gbest) (17, 18). Al mapear este comportamiento sobre la superficie de aptitud (fitness landscape), cualquier configuración que intente asignar microservicios excediendo el umbral físico de un nodo de 2000 MIPS colisiona contra una barrera de costo prohibitivo (W_OVERLOAD). Esto altera abruptamente el gradiente del espacio de búsqueda, rompiendo la inercia de la partícula y redirigiendo su vector de atracción hacia regiones vectoriales asociadas a los nodos subutilizados de alta capacidad (4000 MIPS). Esta equilibrada distribución adaptativa justifica plenamente que las pruebas de ANOVA y t de Welch arrojen una significancia extrema, confirmando que las mejoras en el retardo no constituyen anomalías estocásticas del simulador iFogSim2, sino la estructuración formal de un patrón de asignación óptimo y científicamente validado (20).

Sin embargo, la viabilidad de este modelo se encuentra con un límite teórico crítico al evaluar su frontera de escalabilidad estructural. Si la infraestructura que se está analizando se expandiera hacia un escenario ultra denso compuesto por 1000 nodos Fog, el espacio de búsqueda combinatorio sufriría una explosión dimensional de carácter exponencial (N M, donde N es el número de nodos y M la cantidad de microservicios). En estas condiciones de alta dimensionalidad, el algoritmo PSO tradicional muestra una fuerte tendencia a la convergencia prematura, provocada por una pérdida muy considerable de la diversidad genética en el enjambre, quedando atrapado de forma irreversible en mínimos locales subóptimos.

De la misma forma, el costo computacional requerido para evaluar iterativamente la función de aptitud en una matriz de tal magnitud incrementaría el overhead del sistema de orquestación, superando el umbral de milisegundos exigido para las decisiones en tiempo real de la capa de borde. Para mitigar esta restricción de escalabilidad en entornos masivos, la literatura contemporánea establece que el algoritmo no debe operar de forma aislada, sino que requiere la integración de esquemas previos de zonificación geográfica, de composición jerárquica de la red o la implementación de enfoques híbridos adaptativos (5, 15). El estadístico F de ANOVA exhibe un comportamiento creciente con la carga, corroborando que la relevancia de escoger la

política de integración se incrementa cuando el sistema está bajo carga, y es este rigor estadístico el cual es clave para soportar decisiones de ingeniería basadas en evidencia cuantitativa y poder así superar limitaciones de trabajos previos que no cuentan con comparaciones robustas frente a procesos estáticos; lo cual constituye un aspecto particularmente innovador de esta discusión es el análisis del trade-off entre latencia y energía.

Se considera que la mejora del rendimiento del cálculo va acompañada de un aumento de consumo de energía equivalente. Sin embargo, los resultados indican que PSO puede desacoplar parcialmente estas variables. Mientras que para carga fija la correlación entre ambas medidas se atenúa en la versión optimizada, las dos políticas estáticas que utilizan una constante están ligadas por una dependencia la cual atenta contra la sustentabilidad del sistema. Esta disminución en el consumo de energía, de hasta el 25.16% en comparación con Random bajo carga pesada, es crucial para la factibilidad de despliegues Fog en ambientes con limitaciones de potencia, tales como sensores remotos o dispositivos embebidos acoplados a baterías.

Es fundamental reconocer que el éxito de PSO en este contexto se atribuye a su mecanismo de búsqueda global, que permite navegar un espacio de soluciones complejo y no convexo sin las limitaciones de los métodos basados en gradientes. Al considerar simultáneamente el consumo en idle y busy de los nodos, el algoritmo realiza una poda efectiva de asignaciones que, aunque podrían parecer óptimas en términos de proximidad física, resultarían costosas energéticamente. El uso de iFogSim2 ha sido fundamental para capturar dinámicas de microservicios que simuladores más antiguos solían omitir. La transición hacia infraestructuras 7G y entornos distribuidos heterogéneos exigirá que estas políticas dinámicas se conviertan en el estándar de la industria. De esta forma la robustez demostrada ante variaciones de carga sugiere que PSO puede actuar como un estabilizador del sistema, reduciendo la dispersión de los resultados y garantizando una experiencia de usuario predecible, fluida y altamente satisfactoria para el consumidor.

Estos hallazgos reafirman que la optimización metaheurística en la capa Fog no solo resuelve problemas técnicos de retardo, sino que sustenta una arquitectura de software más resiliente para el futuro del procesamiento descentralizado en entornos académicos de alto nivel tecnológico. La reducción drástica en las violaciones de SLA demuestra que la adaptatividad es una necesidad operativa absoluta para aplicaciones críticas de IoT, tales como el control industrial en tiempo real, donde cada milisegundo de retraso puede comprometer la seguridad de los datos gestionados en el borde. Este salto metodológico permite pensar aplicaciones más ambiciosas en el diseño de redes eficientes e inteligentes a gran escala para el siglo veintiuno, dando un marco de trabajo sólido, confiable y de vanguardia técnica.

CONCLUSIONES

Los resultados obtenidos en esta investigación de carácter empírico confirman de manera fehaciente que la implementación de la política de colocación dinámica basada en Particle Swarm Optimization (PSO) constituye una solución técnica altamente efectiva para optimizar el rendimiento en entornos Fog heterogéneos de manera integral. La disminución coherente de

latencia en todos los casos de prueba, entre un 20% y un 39%, indica que la metaheurística adaptativa es capaz de superar las limitaciones de las soluciones estáticas, como Round-Robin y Random. Esta es una constatación crucial para aplicaciones del Internet de las Cosas con tiempos de respuesta críticos, asegurando que el procesamiento suceda en el nodo más adecuado basado en su capacidad MIPS y carga actual. Y aún más, esta mejora en eficiencia no incrementa el consumo de energía, por el contrario, PSO disminuye en un 14% a 25% el consumo total. Esta doble ventaja en desempeño y en sostenibilidad hace que la optimización de enjambre de partículas sea una herramienta clave para la ingeniería de software en el borde de la red. La verificación estadística proporciona un nivel de confianza superior al 99,9 %, facilitando de esta manera la implementación de infraestructuras resilientes.

En segundo lugar, el trabajo revela que la estabilidad operativa y la escalabilidad son características fundamentales que solo pueden ser aseguradas por políticas adaptativas. En contraste con estrategias estáticas, que experimentaron una fuerte degradación y aumento en la violación de SLA cuando se enfrentan con cargas, PSO pudo ejercer un control riguroso sobre el cumplimiento del QoS. La disminución de violaciones de SLA de unos promedios superiores a 11 en Random a solo 0.13 resalta la relevancia de tener en cuenta la heterogeneidad de recursos. Este comportamiento permite evitar cuellos de botella en nodos críticos y garantiza un throughput mayor, clave para servicios en tiempo real. La investigación indica que la complejidad de los microservicios requiere de una visión holística eficiente. Por tanto, gestionar globalmente consciente de que el sistema necesita recursos es una necesidad para la arquitectura de en IoT ecosistemas presentes y asegurar respuesta exitosa de demanda de picos y minimización de riesgos operacionales potencialmente severos.

Por lo tanto, esta investigación proporciona un fundamento metodológico para la computación de niebla, implicando que el futuro de la orquestación de IoT prospera hacia la agregación de herramientas de simulación avanzada y algoritmos inteligentes de precisión. Se concluye que el desacoplamiento obtenido entre latencia y energía bajo PSO representa un paso hacia la sostenibilidad tecnológica, haciendo que el procesamiento no afecte línea a linea en la huella de carbono. Este estudio confirma la superioridad de las metaheurísticas y puede continuar investigando sobre la movilidad y la seguridad. La correlación entre el éxito de la optimización y la escala de carga identificada indica que las políticas adaptativas serán aún más importantes en futuras infraestructuras 7G. La implementación exitosa de PSO es una bonificación significativa, ya que demuestra que la deuda técnica puede ser mitigada usando inteligencia artificial, lo que permite el desarrollo de un tejido digital más inteligente y autónomo.

REFERENCIAS BIBLIOGRÁFICAS

1. Shi W, Cao J, Zhang Q, Li Y, Xu L. Edge computing: Vision and challenges. IEEE Internet of Things Journal. 2016;3(5):637-646.
2. Bonomi F, Milito R, Zhu J, Addepalli S. Fog computing and its role in the Internet of Things. In: Proc First Ed MCC Workshop Mobile Cloud Computing; 2012. p. 13-16.

3. Skarlat O, Schulte S, Borkowski M, Leitner P. Resource provisioning for IoT services in the fog. In: Proc IEEE SOSE; 2017. p. 32-41.
4. Rejiba Z, et al. A survey on mobility-induced service migration in fog, edge, and related computing paradigms. ACM Computing Surveys. 2019;52(2):Art 37.
5. Mahmud R, Ramamohanarao K, Buyya R. Application placement in fog computing environments: A taxonomy, review and future directions. Journal of Systems and Software. 2018;147:1-24.
6. Gill M, Singh D. A comprehensive study of simulation frameworks and research directions in fog computing. Comput Sci Rev. 2021;40:100391.
7. Gupta H, et al. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. Softw Pract Exp. 2017;47(9):1275-1296.
8. de Aguilar U, Cañizares PC, Núñez A. Simcan2Fog: A discrete-event platform for the modelling and simulation of Fog computing environments. SoftwareX. 2025;32:102424.
9. Andreoli R, Zhao J, Cucinotta T, Buyya R. CloudSim 7G: An Integrated Toolkit for Modeling and Simulation of Future Generation Cloud Computing Environments. Softw Pract Exp. 2025;55(6):1041-1058.
10. Qayyum T, et al. FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment. IEEE Access. 2018;6:63570–63583.
11. Panda SK, Bhagat A. IFCSim: An IoT-Fog-Cloud Simulator for Analyzing Performance of Topologies. In: 15th Int Conf Comput Commun Netw Technol (ICCCNT); 2024.
12. Riley GF, Reddy D. Simulating realistic packet routing without routing protocols. In: Proc Workshop Princ Adv Distrib Simul (PADS); 2005. p. 151–158.
13. Maiti P, Sahoo B, Turuk AK. Low Latency Aware Fog Nodes Placement in Internet of Things Service Infrastructure. J Circuits Syst Comput. 2022;31(1):2250017.
14. Jadhav A, Mini S. Algorithms for task allocation in fog computing. In: 2nd World Conf Commun Comput (WCONF); 2024.
15. Singh S, Vidyarthi DP. A hybrid model using JAYA-GA metaheuristics for placement of fog nodes in fog-integrated cloud. J Ambient Intell Humanized Comput. 2024;15(7):3035–3052.
16. Morabito R, et al. Evaluating performance of containerized IoT services for clustered devices at the network edge. IEEE Internet of Things Journal. 2018;4(4):1019-1030.
17. Kennedy J, Eberhart R. Particle swarm optimization. In: Proc IEEE ICNN; 1995. Vol. 4, p. 1942-1948.
18. Eberhart R, Shi Y. Particle swarm optimization: Developments, applications and resources. In: Proc 2001 Congress Evolutionary Computation; 2001. Vol. 1, p. 81-86.
19. Mahmud R, Ramamohanarao K, Buyya R. iFogSim2: An extended iFogSim simulator for mobility, clustering and microservice management in edge and fog computing environments. J Syst Softw. 2022;190:111351.
20. Mahmud R, Buyya R. Modeling and simulation of fog and edge computing environments using ifogsim toolkit. In: Fog and Edge Computing. wiley; 2019. p. 433–165

ANEXO CÓDIGO DEL ALGORITMO

```
package org.fog.examples;

import org.cloudbus.cloudsim.*;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.power.PowerHost;
import org.cloudbus.cloudsim.power.models.PowerModel;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;
import org.cloudbus.cloudsim.sdn.overbooking.BwProvisionerOverbooking;
import org.cloudbus.cloudsim.sdn.overbooking.PeProvisionerOverbooking;

import org.fog.application.AppEdge;
import org.fog.application.AppLoop;
import org.fog.application.AppModule;
import org.fog.application.Application;
import org.fog.entities.Actuator;
import org.fog.entities.FogDevice;
import org.fog.entities.FogDeviceCharacteristics;
import org.fog.entities.Sensor;
import org.fog.placement.Controller;
import org.fog.placement.ModulePlacement;
import org.fog.policy.AppModuleAllocationPolicy;
import org.fog.scheduler.StreamOperatorScheduler;
import org.fog.utils.Config;
import org.fog.utils.FogLinearPowerModel;
import org.fog.utils.FogUtils;
import org.fog.utils.Logger;

import java.util.*;

public class PSOPlacement {

    /* ---------------------------
       1) PSO DATA STRUCTURES
       --------------------------- */
    static class Particle {
        double() position;      // discrete: device index per module
        double() velocity;
        double fitness;

        double() bestPosition;
        double bestFitness = Double.MAX_VALUE;
    }

    /* ---------------------------
```

```
   2) PSO HYPERPARAMETERS
   ------------------------- */
private static final int SWARM_SIZE = 20;
private static final int MAX_ITERATIONS = 60;

private static final double W  = 0.72;
private static final double C1 = 1.49;
private static final double C2 = 1.49;

/* -------------------------
   3) EXPERIMENT SETTINGS
   ------------------------- */
private static final long SEED = 42L;
private static final Random RNG = new Random(SEED);

// Objective weights (paper-friendly)
private static final double W_LAT = 1.0;          // latency term
private static final double W_OVERLOAD = 1_000_000; // hard penalty
private static final double W_UTIL = 10.0;        // soft load-balancing

/* -------------------------
   4) IFOGSIM ENTITIES
   ------------------------- */
private static List<FogDevice> fogDevices = new ArrayList<>();
private static Application application;

private static final List<String> moduleNames = new ArrayList<>();
private static Particle globalBest;

// Latency map for fitness: "minId-maxId" -> ms
private static final Map<String, Double> latencyMs = new HashMap<>();

// Capacity table to avoid API differences
private static final Map<Integer, Integer> capMips = new HashMap<>();

// Baseline mapping (for comparison in console)
private static Map<String, Integer> baselinePlacement;

/* ============================
   MAIN
   ============================ */
public static void main(String() args) {
   try {
      Logger.ENABLED = true;

      System.out.println("=== PSO Fog Placement Experiment (NetBeans-safe) ===");
```

```
        System.out.println("Seed=" + SEED + " | swarm=" + SWARM_SIZE + " | iter=" +
MAX_ITERATIONS);

        CloudSim.init(1, Calendar.getInstance(), false);

        fogDevices = createFogDevices();
        application = createApplication("PSOApp");

        moduleNames.clear();
        for (AppModule m : application.getModules()) moduleNames.add(m.getName());

        // Baseline: place all modules on Fog-1 (first fog node) to compare
        baselinePlacement = new HashMap<>();
        for (String mn : moduleNames) baselinePlacement.put(mn, fogDevices.get(0).getId());

        double baselineFitness = evaluateFitnessFromMap(baselinePlacement);

        // PSO optimization
        runPSO();

        // Convert best solution to placement map
        Map<String, Integer> bestMap =
convertPositionToModuleDeviceMap(globalBest.bestPosition);
        double bestFitness = evaluateFitnessFromMap(bestMap);

        // Build placement policy for iFogSim execution
        Map<String, List<Integer>> placementForIFogSim =
convertToModuleDeviceListMap(bestMap);
        ModulePlacement policy = new PSOModulePlacement(fogDevices, application,
placementForIFogSim);

        // Controller signature in stable forks
        List<Sensor> sensors = new ArrayList<>();
        List<Actuator> actuators = new ArrayList<>();
        Controller controller = new Controller("master-controller", fogDevices, sensors,
actuators);

        // Inject mapping
        controller.submitApplication(application, policy);

        // Run simulation (even with empty sensors/actuators, iFogSim will execute basic
setup)
        CloudSim.startSimulation();
        CloudSim.stopSimulation();

        // Report
        System.out.println("\n=== RESULTS (Synthetic experiment, topology fixed) ===");
```

```
        System.out.println("Baseline fitness: " + String.format("%.6f", baselineFitness));
        System.out.println("PSO best fitness: " + String.format("%.6f", bestFitness));
        System.out.println("\nBaseline placement:");
        printPlacement(baselinePlacement);
        System.out.println("\nPSO placement:");
        printPlacement(bestMap);

        System.out.println("\nDone.");

    } catch (Throwable t) {
        t.printStackTrace();
        Log.printLine("Unwanted errors happen");
    }
}

/* ============================
   TOPOLOGY CREATION
   ============================ */
private static List<FogDevice> createFogDevices() throws Exception {
    List<FogDevice> devices = new ArrayList<>();

    // Three heterogeneous Fog nodes
    FogDevice fog1 = createFogDevice("Fog-1", 4000, 4096, 10000, 10000, 0, 0.0, 0.0);
    FogDevice fog2 = createFogDevice("Fog-2", 2500, 4096, 10000, 10000, 1, 10.0, 0.0);
    FogDevice fog3 = createFogDevice("Fog-3", 1800, 4096, 10000, 10000, 1, 15.0, 0.0);

    // Parent-child hierarchy
    fog2.setParentId(fog1.getId());
    fog3.setParentId(fog1.getId());

    fog1.addChild(fog2.getId());
    fog1.addChild(fog3.getId());

    // IMPORTANT: used by FogDevice.sendDownFreeLink in many forks
    fog1.getChildToLatencyMap().put(fog2.getId(), 10.0);
    fog1.getChildToLatencyMap().put(fog3.getId(), 15.0);

    // Levels
    fog1.setLevel(0);
    fog2.setLevel(1);
    fog3.setLevel(1);

    devices.add(fog1);
    devices.add(fog2);
    devices.add(fog3);

    // Capacity table (for fitness)
```

```
    capMips.put(fog1.getId(), 4000);
    capMips.put(fog2.getId(), 2500);
    capMips.put(fog3.getId(), 1800);

    // Pairwise latencies for PSO fitness (ms)
    putLatency(fog1.getId(), fog2.getId(), 10.0);
    putLatency(fog1.getId(), fog3.getId(), 15.0);
    putLatency(fog2.getId(), fog3.getId(), 25.0);

    return devices;
}

private static void putLatency(int idA, int idB, double ms) {
    int a = Math.min(idA, idB);
    int b = Math.max(idA, idB);
    latencyMs.put(a + "-" + b, ms);
}

private static double getLatency(int idA, int idB) {
    if (idA == idB) return 0.0;
    int a = Math.min(idA, idB);
    int b = Math.max(idA, idB);
    return latencyMs.getOrDefault(a + "-" + b, 50.0);
}

/**
 * Canonical FogDevice builder (works across iFogSim2 forks that include these packages).
 */
private static FogDevice createFogDevice(String name,
                                         long mips,
                                         int ram,
                                         double upBw,
                                         double downBw,
                                         int level,
                                         double upLatency,
                                         double ratePerMips) throws Exception {

    // PEs
    List<Pe> peList = new ArrayList<>();
    peList.add(new Pe(0, new PeProvisionerOverbooking(mips)));

    // Host
    int hostId = FogUtils.generateEntityId();
    long storage = 1_000_000;
    int bw = 10_000;

    PowerModel pm = new FogLinearPowerModel(0.01, 0.001);
```

```
    PowerHost host = new PowerHost(
            hostId,
            new RamProvisionerSimple(ram),
            new BwProvisionerOverbooking(bw),
            storage,
            peList,
            new StreamOperatorScheduler(peList),
            pm
    );

    List<Host> hostList = new ArrayList<>();
    hostList.add(host);

    VmAllocationPolicy vmPolicy = new AppModuleAllocationPolicy(hostList);

    // Characteristics
    String arch = Config.FOG_DEVICE_ARCH;
    String os = Config.FOG_DEVICE_OS;
    String vmm = Config.FOG_DEVICE_VMM;
    double timeZone = Config.FOG_DEVICE_TIMEZONE;
    double cost = Config.FOG_DEVICE_COST;
    double costPerMem = Config.FOG_DEVICE_COST_PER_MEMORY;
    double costPerStorage = Config.FOG_DEVICE_COST_PER_STORAGE;
    double costPerBw = Config.FOG_DEVICE_COST_PER_BW;

    FogDeviceCharacteristics ch = new FogDeviceCharacteristics(
            arch, os, vmm, host, timeZone, cost, costPerMem, costPerStorage, costPerBw
    );

    double schedulingInterval = 10.0;

    FogDevice d = new FogDevice(
            name,
            ch,
            vmPolicy,
            new LinkedList<Storage>(),
            schedulingInterval,
            upBw,
            downBw,
            upLatency,
            ratePerMips
    );

    d.setLevel(level);
    return d;
}
```

```
/* ============================
   APPLICATION CREATION
   ============================ */
private static Application createApplication(String appId) {
    Application app = Application.createApplication(appId, FogUtils.generateUniqueId());

    // Modules chosen to stress heterogeneous capacities (paper hypothesis)
    AppModule m1 = new AppModule("m1", 1600, 100, 100, 100);
    AppModule m2 = new AppModule("m2", 1200, 100, 100, 100);
    app.addAppModule(m1);
    app.addAppModule(m2);

    // Module-to-module edge affects latency term of objective
    app.addAppEdge("m1", "m2", 1000, 500, "m2_input", AppEdge.MODULE);

    // Optional edges (kept but do not require real sensors/actuators for this example)
    app.addAppEdge("sensor", "m1", 1000, 500, "sensor_type", AppEdge.SENSOR);
    app.addAppEdge("m2", "actuator", 1000, 500, "actuator_type", AppEdge.ACTUATOR);

    app.addAppLoop(new AppLoop(new ArrayList<String>() {{
        add("sensor"); add("m1"); add("m2"); add("actuator");
    }}));

    return app;
}

/* ============================
   PSO CORE
   ============================ */
private static void runPSO() {
    Particle() swarm = new Particle(SWARM_SIZE);
    globalBest = new Particle();
    globalBest.bestFitness = Double.MAX_VALUE;

    // Init swarm
    for (int i = 0; i < SWARM_SIZE; i++) {
        swarm(i) = initParticle(moduleNames.size(), fogDevices.size());
        swarm(i).fitness = evaluateFitness(swarm(i).position);
        updatePersonalBest(swarm(i));
        updateGlobalBest(swarm(i));
    }

    // Iterate
    for (int iter = 0; iter < MAX_ITERATIONS; iter++) {
        for (Particle p : swarm) {
            updateVelocity(p);
            updatePosition(p);
```

```
            p.fitness = evaluateFitness(p.position);
            updatePersonalBest(p);
            updateGlobalBest(p);
        }
        if ((iter + 1) % 10 == 0) {
            System.out.println("Iter " + (iter + 1) + "/" + MAX_ITERATIONS +
                    " | bestFitness=" + String.format("%.6f", globalBest.bestFitness));
        }
    }
}

private static Particle initParticle(int moduleCount, int deviceCount) {
    Particle p = new Particle();
    p.position = new double(moduleCount);
    p.velocity = new double(moduleCount);

    for (int i = 0; i < moduleCount; i++) {
        p.position(i) = RNG.nextInt(deviceCount);
        p.velocity(i) = 0.0;
    }
    p.bestPosition = p.position.clone();
    return p;
}

private static void updateVelocity(Particle p) {
    for (int i = 0; i < p.position.length; i++) {
        double r1 = RNG.nextDouble();
        double r2 = RNG.nextDouble();
        double cognitive = C1 * r1 * (p.bestPosition(i) - p.position(i));
        double social = C2 * r2 * (globalBest.bestPosition(i) - p.position(i));
        p.velocity(i) = W * p.velocity(i) + cognitive + social;
    }
}

private static void updatePosition(Particle p) {
    for (int i = 0; i < p.position.length; i++) {
        p.position(i) += p.velocity(i);
        p.position(i) = Math.round(p.position(i));

        if (p.position(i) < 0) p.position(i) = 0;
        if (p.position(i) >= fogDevices.size()) p.position(i) = fogDevices.size() - 1;
    }
}

private static void updatePersonalBest(Particle p) {
    if (p.fitness < p.bestFitness) {
        p.bestFitness = p.fitness;
```

```
            p.bestPosition = p.position.clone();
        }
    }

    private static void updateGlobalBest(Particle p) {
        if (p.bestFitness < globalBest.bestFitness) {
            globalBest.bestFitness = p.bestFitness;
            globalBest.bestPosition = p.bestPosition.clone();
        }
    }

    /* ============================
       FITNESS (Hypothesis-oriented)
       ============================ */

    private static double evaluateFitness(double() position) {
        Map<String, Integer> map = convertPositionToModuleDeviceMap(position);
        return evaluateFitnessFromMap(map);
    }

    /**
     * Fitness = latency(m1->m2) + overloadPenalty + utilizationPenalty
     * - latency: encourages co-location or low-latency links
     * - overload: hard constraint (capacity exceed -> huge penalty)
     * - utilization: soft balancing (prefers spreading load when possible)
     */
    private static double evaluateFitnessFromMap(Map<String, Integer> placement) {

        // Load per device
        Map<Integer, Integer> load = new HashMap<>();
        for (AppModule m : application.getModules()) {
            int devId = placement.get(m.getName());
            load.put(devId, load.getOrDefault(devId, 0) + m.getMips());
        }

        // Overload penalty
        double overloadPenalty = 0.0;
        double utilPenalty = 0.0;

        for (FogDevice d : fogDevices) {
            int devId = d.getId();
            int l = load.getOrDefault(devId, 0);
            int cap = capMips.getOrDefault(devId, 2000);

            if (l > cap) overloadPenalty += (l - cap);

            // utilization penalty: (load/cap)^2 to discourage hotspot
```

```
            double u = (cap > 0) ? ((double) l / cap) : 10.0;
            utilPenalty += (u * u);
        }

        // Latency term: only module-to-module edges
        double lat = 0.0;
        for (AppEdge e : application.getEdges()) {
            if (e.getEdgeType() == AppEdge.MODULE) {
                Integer s = placement.get(e.getSource());
                Integer t = placement.get(e.getDestination());
                if (s != null && t != null) lat += getLatency(s, t);
            }
        }

        return (W_LAT * lat) + (W_OVERLOAD * overloadPenalty) + (W_UTIL * utilPenalty);
    }

    /* ============================
       CONVERSIONS
       ============================ */

    private static Map<String, Integer> convertPositionToModuleDeviceMap(double() position)
{
        Map<String, Integer> placement = new HashMap<>();
        for (int i = 0; i < position.length; i++) {
            String moduleName = moduleNames.get(i);
            int deviceIndex = (int) Math.round(position(i));
            if (deviceIndex < 0) deviceIndex = 0;
            if (deviceIndex >= fogDevices.size()) deviceIndex = fogDevices.size() - 1;
            placement.put(moduleName, fogDevices.get(deviceIndex).getId());
        }
        return placement;
    }

    private static Map<String, List<Integer>> convertToModuleDeviceListMap(Map<String,
Integer> simpleMap) {
        Map<String, List<Integer>> out = new HashMap<>();
        for (Map.Entry<String, Integer> e : simpleMap.entrySet()) {
            out.put(e.getKey(), Collections.singletonList(e.getValue()));
        }
        return out;
    }

    private static void printPlacement(Map<String, Integer> placement) {
        for (String m : moduleNames) {
            int devId = placement.get(m);
```

```
            System.out.println("  " + m + " -> " + fogDeviceNameById(devId) + " (id=" + devId +
")");
        }
    }

    private static String fogDeviceNameById(int id) {
        for (FogDevice d : fogDevices) {
            if (d.getId() == id) return d.getName();
        }
        return "UnknownDevice";
    }

    /* ============================
       PSO MODULE PLACEMENT POLICY
       ============================ */

    public static class PSOModulePlacement extends ModulePlacement {
        private final Map<String, List<Integer>> psoPlacementMap;

        public PSOModulePlacement(List<FogDevice> fogDevices,
                                  Application application,
                                  Map<String, List<Integer>> psoPlacementMap) {
            this.setFogDevices(fogDevices);
            this.setApplication(application);
            this.psoPlacementMap = psoPlacementMap;

            this.setModuleToDeviceMap(new HashMap<>());
            this.setDeviceToModuleMap(new HashMap<>());
        }

        @Override
        protected void mapModules() {
            for (Map.Entry<String, List<Integer>> e : psoPlacementMap.entrySet()) {
                String moduleName = e.getKey();
                int deviceId = e.getValue().get(0);

                getModuleToDeviceMap().put(
                        getApplication().getModuleByName(moduleName),
                        getFogDeviceById(deviceId)
                );

                List<AppModule> placed = getDeviceToModuleMap().getOrDefault(deviceId, new
ArrayList<>());
                placed.add(getApplication().getModuleByName(moduleName));
                getDeviceToModuleMap().put(deviceId, placed);
            }
        }
```

```
    }
}
```