STATISTICS, OPTIMIZATION AND INFORMATION COMPUTING
Stat., Optim. Inf. Comput., Vol. **x**, Month 202x, pp 0–43.
Published online in International Academic Press (www.IAPress.org)

Latency-Cost-Requirement-Aware Population Initialization for Metaheuristic Cloudlet Placement in Edge Computing

Heba El-Rahman Hassan[1,*], Hassan Al-Mahdi[1], Hamed Nassar[1], and Mohamed Tahoun[1,2]

[1] *Computer Science Department, Faculty of Computers and Informatics, Suez Canal University, 41522 Ismailia, Egypt*
[2] *Department of Cybersecurity, College of Engineering and Information Technology, Buraydah Private Colleges, Buraydah 51418, Kingdom of Saudi Arabia*

Abstract Cloudlet placement represents a critical design consideration in mobile edge computing because placement decisions influence service delay, deployment cost, resource utilization, and workload distribution. The resulting Cloudlet Placement Problem (CPP) is a complex optimization problem that involves discrete placement decisions, heterogeneous cloudlet capacities, varying end-device requirements, and multiple competing objectives. Although metaheuristic approaches are widely used to address the combinatorial nature of the CPP, purely random population initialization may generate solutions with high deployment cost, excessive latency, poor resource balance, or residual infeasibility. As a result, a substantial portion of the search process may be spent improving low-quality initial placements. This paper proposes Latency-Cost-Requirement-Aware (LCRA), a CPP-specific, problem-aware population-initialization framework evaluated with genetic algorithm (GA) and particle swarm optimization (PSO) update mechanisms. LCRA constructs the initial population using problem-specific indicators that capture deployment cost, user-to-cloudlet latency, coverage capability, and workload distribution. A preference parameter controls the initialization strategy by prioritizing latency minimization, deployment-cost reduction, or a balanced trade-off between the two objectives. Extensive experiments are conducted on datasets from five New York City urban regions to evaluate the proposed framework in terms of end-device-to-cloudlet latency, deployment cost, and runtime. The results show that, compared with the corresponding random-initialization baselines, LCRA reduces the average deployment cost by 9.19–12.60% under the cost-focused setting and the average total latency by 0.22–0.36% under the latency-focused setting. Compared with the evaluated Latency-Modified Approximate Cloudlet Placement (LMACP) implementation, LCRA's reported deployment-cost and average total latency values are lower by up to 36.21% and 1.82%, respectively. In addition, all evaluated LCRA runs passed the implemented final feasibility checks, indicating complete end-device coverage and zero residual constraint violations under the repair-aware evaluation.

Keywords Cloudlet Placement Problem, Edge Computing, Mobile Cloud Computing, Problem-Aware Initialization, Search-Space Seeding, Metaheuristic Optimization, Cost Optimization, Latency Reduction, Load Balancing.

DOI: 10.19139/soic-2310-5070-5047

1. Introduction

The rapid growth of mobile and Internet of Things (IoT) applications has increased the demand for low-latency, reliable, and real-time computing services. Applications such as augmented reality, online gaming, vehicular communication, remote surgery, and industrial IoT cannot always rely on distant centralized cloud data centers because long communication paths may introduce unacceptable delays. Mobile Edge Computing (MEC) addresses this limitation by deploying computing resources closer to end users through cloudlets located at selected network sites, such as base stations, access points, and edge aggregation nodes [1], [2]. As illustrated in Fig. 1, MEC establishes an intermediate computing layer between end devices and the remote cloud.

*Correspondence to: Heba El-Rahman Hassan (e-mail: heba_hassan@ci.suez.edu.eg), Computer Science Department, Faculty of Computers and Informatics, Suez Canal University, 41522 Ismailia, Egypt.

ISSN 2310-5070 (online) ISSN 2311-004X (print)
Copyright © 202x International Academic Press

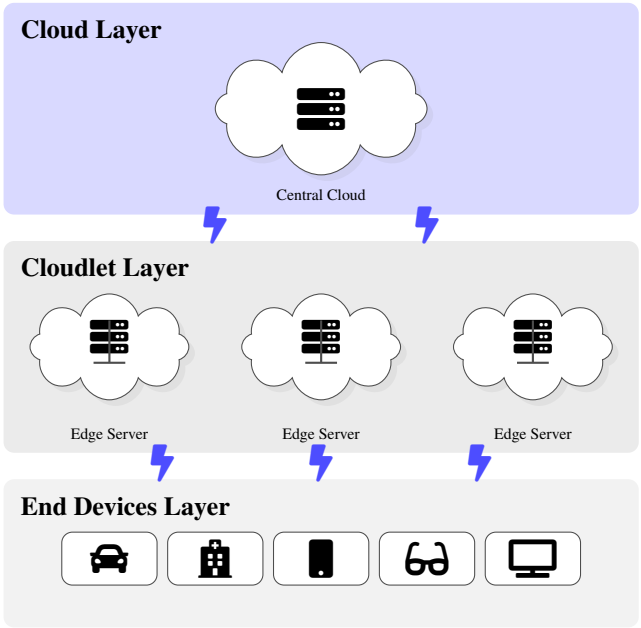


Figure 1. Mobile Edge Computing Architecture.

The effectiveness of MEC depends strongly on cloudlet placement. Deploying cloudlets far from dense user regions may increase latency, whereas deploying unnecessary high-capacity cloudlets increases operational cost. Concentrating workloads on a limited number of cloudlets may also cause resource imbalance and degrade service quality. The Cloudlet Placement Problem (CPP) therefore determines suitable locations for heterogeneous cloudlets serving heterogeneous end devices while considering deployment cost, communication latency, coverage, and resource-capacity constraints [3], [4]. Because the CPP combines discrete location decisions, heterogeneous capacities, diverse device demands, and conflicting objectives, it is NP-hard, making exhaustive search impractical for large edge networks [5].

Exact formulations, including mixed-integer linear programming models, provide rigorous CPP representations and are useful for small instances and benchmark analysis [5]. However, their computational cost increases rapidly with the numbers of devices, candidate points, and cloudlets. Problem-specific heuristics reduce this burden but may not generalize across different deployment conditions. Population-based metaheuristics, including genetic algorithms (GA) and particle swarm optimization (PSO), provide flexible alternatives for large and complex CPP instances [6]. However, purely random placement selects starting cloudlet locations without explicitly using the relationships among communication range, resource capacity, and device demand. A placement may cover many devices geographically yet lack the capacity to serve them, while a capacity-feasible placement may still incur unnecessary deployment cost or latency. Assignment and repair can reduce infeasibility, but do not by themselves ensure that the starting population reflects the selected cost–latency preference. This motivates examining population initialization as a distinct design stage, with its contribution evaluated under unchanged GA or PSO update mechanisms.

To address this limitation, this paper proposes a problem-aware population-initialization framework that improves the initial search space before iterative optimization begins. Instead of relying exclusively on random placement, the framework constructs candidate solutions using information related to deployment cost, user-to-cloudlet latency, coverage potential, workload balance, and resource requirements. The initialization strategy is selected according to the cost-latency preference, while a random component preserves population diversity. The resulting candidates are subsequently processed through common assignment, repair, fitness-evaluation, and GA- or PSO-based update mechanisms.

This paper makes four principal contributions. First, Latency-Cost-Requirement-Aware (LCRA) is proposed as a CPP-specific population-initialization framework. Its contribution is the coordinated use of latency-oriented, balanced cost–latency, cost-focused, and random seeding under the selected cost–latency preference, rather than the general concept of informed initialization. The framework provides informed initial populations for the GA and PSO update mechanisms evaluated in this study. Second, a common feasibility-aware placement pipeline integrates Priority-Based Device Assignment (PBDA), Coverage and Resource Repair (CRR), resource-capacity verification, residual constraint-violation penalties, and selectable GA or PSO-based population updates. Third, Latency-Modified Approximate Cloudlet Placement (LMACP) is established as an external experimental baseline by retaining the supplied Approximate Cloudlet Placement (ACP) procedure [7] and replacing its distance-based latency calculation with the four-component service-latency model used in this study. Finally, the framework is evaluated on real-world cloudlet-placement instances constructed from New York City Wi-Fi hotspot data across five boroughs under different cost-latency preferences. LCRA-GA and LCRA-PSO are compared with their random-initialization counterparts and with the LMACP baseline.

The remainder of this paper is organized as follows. Section 2 reviews related cloudlet-placement and metaheuristic approaches. Section 3 presents the system, cost, and latency models. Section 4 formulates the multi-objective CPP and its constraints. Section 5 describes the proposed LCRA framework, including initialization, device assignment, repair, fitness evaluation, and GA/PSO population updates. Section 6 presents the experimental setup, Section 7 discusses the results, and Section 8 concludes the paper.

2. Related Work

Cloudlet and edge-server placement significantly affects latency, deployment cost, coverage, workload distribution, resource efficiency, energy consumption, and quality of service [8, 9]. The CPP selects suitable cloudlet locations near users, Internet of Things devices, mobile terminals, or access points while balancing these conflicting objectives. Because CPP is generally formulated as an NP-hard multi-objective combinatorial optimization problem [5, 7, 10], existing studies employ mathematical programming, approximation algorithms, heuristics, learning-based techniques, and metaheuristic algorithms [8, 9, 11].

2.1. Exact and Approximation-Based Placement Methods

Exact and approximation-based methods provide rigorous CPP models and theoretical guarantees. Bhatta and Mashayekhy [12] formulated a generalized cost-aware CPP for vehicular edge environments, while Yang et al. [5] jointly modelled cloudlet placement and task allocation as an NP-hard mixed-integer linear program. Bahreini and Grosu [13] developed matching- and greedy-based algorithms for multi-component application placement, and Bhatta and Mashayekhy [7] proposed a bifactor approximation algorithm with separate guarantees for latency and placement cost while ensuring device coverage. Related optimization studies address service caching, cost-aware service placement, digital-twin replica placement, model-instance deployment, and edge-cloud resource provisioning using exact, approximation, randomized, and heuristic techniques [14–20]. Although these approaches provide strong theoretical baselines, their scalability is limited, and they do not address population initialization for metaheuristic search.

2.2. Heuristic and Metaheuristic Placement Methods

Heuristic and metaheuristic methods improve scalability by producing high-quality solutions within practical computational time. GA, PSO, ACO, GWO, and hybrid evolutionary methods have been applied to placement, resource allocation, offloading, and scheduling [9, 11]. Representative approaches include PSO–GA workflow-aware cloudlet placement [6], multi-objective GA for service placement and load distribution [21], and hybrid or learning-assisted evolutionary methods [22–27]. However, these studies primarily emphasize search operators, hybridization, encoding, or objective-function design rather than initial search-space construction.

Reinforcement learning, deep Q-learning, and multi-agent learning have also been used for adaptive placement [28–31]. Other studies consider QoE, workload variation, mobility, fidelity, throughput, server selection, task

processing, and system robustness [19, 20, 32–36]. Although these approaches improve placement realism, they may require substantial training and generally do not treat initialization as an independent design problem.

Related fog-computing research also addresses latency through task scheduling. Rani et al. [37] propose a Quality of Service Priority Tuple Scheduling (QoSPTS) approach and evaluate latency, network utilization, energy consumption, and execution cost using iFogSim. This scheduling focus differs from constructing initial populations for cloudlet-placement optimization.

2.3. Initialization and Search-Space Construction

Initialization approaches differ in the information used to construct candidates and in their relationship to the subsequent optimizer. Li et al. [38] propose diagonal linear uniform initialization based on diagonal-subspace sampling and evaluate it with differential evolution, PSO, cuckoo search, and artificial bee colony algorithms. Their work addresses numerical search-space sampling rather than cloudlet-specific placement construction. Maia et al. [21] combine random and heuristic initial solutions within a multi-objective GA for service placement and load distribution, considering response-deadline violations, operational cost, and service availability.

Abdelhaliem et al. [39] incorporate refracted opposition-based learning into the initialization of an enhanced binary Secretary Bird Optimization Algorithm for feature selection. Their approach generates alternative candidates using variable bounds and a refraction parameter, followed by conversion into binary feature selections. This diversifies the starting population through search-space transformations, whereas LCRA constructs initial placements using CPP-specific cost, latency, coverage, and resource information.

Other placement studies use problem information within their solution-construction procedures. Zhang et al.'s workflow-aware method [6] combines PSO, GA operators, and an encoding library that prevents conflicting cloudlet locations. The graph-clustering method of Zhang et al. [3] instead combines a graph convolutional network with differentiable K-means to determine server locations and assignments. Recursive clustering [40], ACO-based placement [41], and resilient clustering and assignment [42] provide further examples of problem-guided construction. These procedures should be distinguished from an independently evaluated population-initialization layer.

Table 1 compares the reviewed approaches using five criteria: information used during initialization or construction, feasibility handling, diversity mechanisms, dependence on a particular optimizer, and supported optimization objectives. This separates the construction of starting solutions from subsequent search operators and distinguishes optimization objectives from evaluation measures such as runtime and feasibility success rate.

The reviewed studies provide foundations for informed initialization and problem-guided placement. The research gap addressed here is how to coordinate cost–latency preferences, heterogeneous resource requirements, feasibility handling, and population diversity within a distinct CPP initialization stage, while separating its contribution from subsequent search operators. LCRA addresses this gap through preference-dependent seeding, resource-aware assignment, and repair within a common framework. Its initialization contribution is evaluated against random initialization while keeping the corresponding GA or PSO update mechanism unchanged.

3. System Model

We consider a heterogeneous cloudlet placement environment with three main entities: candidate points, cloudlets, and end devices. Let $\mathcal{P} = \{p_k\}_{k=1}^{M}$ denote the set of candidate points, where each p_k is defined by coordinates (x_{p_k}, y_{p_k}). Candidate points are predefined feasible locations, such as base stations, access points, or edge aggregation sites, where cloudlets may be deployed. These locations are assumed to satisfy basic requirements, including physical accessibility, power availability, and network connectivity.

Each sampled instance is evaluated as a static placement snapshot: device locations, communication conditions, and workload parameters remain fixed during a placement run.

Let $\mathcal{C} = \{c_j\}_{j=1}^{L}$ denote the set of cloudlets. Each cloudlet $c_j \in \mathcal{C}$ is characterized by a resource specification tuple $\mathcal{R}_{c_j} = \{ P_{c_j}, m_{c_j}, s_{c_j}, r_{c_j}, \tau_{c_j} \}$, where, P_{c_j} denotes the effective processor capacity (e.g., aggregated CPU throughput in GHz), m_{c_j} the main-memory capacity in GB, s_{c_j} the persistent storage capacity in GB, and $r_{c_j} > 0$

Table 1. Comparison of initialization and solution-construction approaches relevant to cloudlet and edge placement.

Study	Method and construction role	Initialization/ construction information	Feasibility handling	Diversity mechanism	Optimizer dependence	Supported objectives
Bhatta and Mashayekhy [7]; Shao et al. [10]	Bifactor approximation; Benders-based placement	Deployment costs, latency, coverage and capacity; user/load distribution	LP rounding and augmentation; constrained Benders decomposition	N/A: nonpopulation procedures	Mathematical-programming and approximation procedures	Deployment/ construction cost and latency/access-delay cost
Xu et al. [14]; Liu et al. [15]	Collaborative service caching; utility-guided cache placement	VM demand/resources and costs; cache popularity, validity and replacement rate	ILP/rounding for resource allocation; cache-placement and exchange procedures	Tabu-search diversification in Liu et al. (search stage)	Game/rounding and tabu-search procedures	Social cost; comprehensive cache utility and associated costs
Li et al. [38]	Diagonal linear uniform initializer	Variable bounds, dimensionality and population size	Bounded numerical search domain; not a CPP constraint model	Uniform diagonal-subspace sampling (initialization)	Tested with DE, PSO, CS, ABC and MOEA/D	Single- and multi-objective numerical benchmarks
Maia et al. [21]	Heuristic-seeded multi-objective GA for service placement	Service-placement/load-distribution information; random and heuristic solutions	MINLP/MILP service-placement formulation under resource limitations	Mixed random and heuristic initial solutions	GA-based service-placement procedure	Response-deadline violations, operational cost and service availability
Zhang et al. [6]	PGEL: PSO with GA operators and encoding library	AP location encodings and workflow transmission/execution times	Encoding library maintains distinct AP placements	Library-based initial encodings; GA variation (search stage)	PGEL's PSO–GA update procedure	Workflow execution time
Luo et al. [28]	DQN-based placement solver	Server locations, access delay and workload-based state/reward	Placement represented through MDP states and movement actions	ϵ-greedy exploration (learning stage)	DQN-ESPA learning procedure	Access delay and workload balance
Zhang et al. [3]	GCN with differentiable K-means placement	BS distances, coverage, channels and workload	Fixed server count through rounding; nearest-server assignment	N/A: graph clustering, not a population initializer	GCN/clustering procedure	Average transmission delay and load balance
Zarei et al. [41]	ACO placement with load-distribution heuristics	BS workload, heterogeneous servers and request deadlines	MINLP formulation; server-selection and request-scheduling heuristics	Ant-colony candidate construction (search stage)	ACO and load-distribution procedures	Load balance, request response time and deadline satisfaction
Sanaei Asl and Hashemi [42]	K-medoids and resilient assignment	BS traffic, demand clusters and server-failure requirements	Capacity, latency and SLA constraints; primary/backup assignment	N/A: clustering/assignment, not population initialization	Two-stage clustering/assignment procedure	Access delay, workload balance and fault tolerance
Sha et al. [32]	QoE-aware integer programming and enhanced GA	QoE/QoS relationship and placement model	Integer-programming and constrained evolutionary placement formulations	Adaptive crossover (search stage)	EESP-EGA; exact solver for small instances	QoE-aware placement
Aljobory and Yazici [36]	Server selection with round-robin task processing	Task size/CPU cycles, mobility and estimated completion delay	Comparison of offloading and local-processing delays	N/A: task/server selection, not population initialization	Selection controller and round-robin service	Task sojourn time (selection criterion, not a placement objective)
Si et al. [26]	Alternating placement and offloading optimization	Candidate BS indices and placement-conditioned offloading decisions	Bounded integer encoding; cloud fallback during fitness evaluation	Random initialization; crossover/mutation (search stage)	PSO for placement; GA for offloading	Average system processing latency
Xing et al. [27]	COA-based placement/offloading solver	Joint placement/offloading encoding and processing-delay model	Boundary correction for coverage and atomicity constraints	Exploration/exploitation phases (search stage)	COA update procedure	Average request processing delay
LCRA (present work)	Preference-dependent LACI/BCLI/CFI and random seeding	Deployment cost, static latency, coverage, workload balance and resource requirements	PBDA assignment, CRR repair and residual-violation checks	Structured/random mixtures, shuffled orders and duplicate-placement rejection	Common initialization pipeline evaluated with GA and PSO	Deployment cost and total service latency; preference-dependent scalarization

N/A denotes that a population-diversity criterion does not apply to the reported nonpopulation procedure; it does not imply an absence of randomness or search exploration. Search-stage and learning-stage mechanisms are identified separately from initialization. The feasibility column describes reported formulations or handling procedures, not unconditional feasibility guarantees.

denotes the nominal communication radius that induces a coverage region for serving nearby end devices. The categorical attribute τ_{c_j} provides specifications of cloudlets (e.g., small, medium or large), often derived from thresholding functions over $(P_{c_j}, m_{c_j}, s_{c_j}, r_{c_j})$. Unless stated otherwise, all capacities are strictly positive and measured in the indicated units.

Let $\mathcal{E} = \{e_i\}_{i=1}^{N}$ denote the set of end devices responsible for generating computational tasks. Each end device $e_i \in \mathcal{E}$ is characterized by its geographical coordinates (x_{e_i}, y_{e_i}) and its corresponding computational task $\mathcal{R}_{e_i} = \{P_{e_i}, m_{e_i}, s_{e_i}, \tau_{e_i}\}$, where P_{e_i}, m_{e_i}, s_{e_i},τ_{e_i} denote the processor demand (in GHz), memory demand (in GB), storage demand (in GB), and task size (in KB), respectively. These parameters collectively describe the computational workload that must be offloaded and processed by a suitable cloudlet within the system.

These system entities and parameters provide the basis for the proposed problem-aware search-space seeding strategy. In particular, candidate-point coordinates, deployment costs, cloudlet capacities, device demands, and latency components are later used to construct initial placement solutions that are more informed than purely random placements. Table 2 summarizes the notation used to describe the system entities, resource attributes, latency terms, assignment variables, and initialization components.

3.1. The Deployment Cost

The deployment of cloudlet c_j at candidate point p_k incurs a cost C_{jk} that includes infrastructure setup $C_{S_{jk}}$, ongoing operational expenditure $C_{O_{jk}}$, corrective maintenance $C_{M_{jk}}$, and location-dependent components $C_{L_{jk}}$, such as real-estate leasing or acquisition, network connectivity, and backhaul. Deployment costs are expressed in virtual units, denoted by V_u. The numerical value assigned to each deployment-cost component is user-defined and may be adjusted according to the considered scenario. The unit V_u is an abstract, scenario-specific cost unit and does not represent a fixed currency, monetary amount, or conversion to any real-world currency. Hence, the value of C_{jk} can be written as

$$C_{jk} = C_{S_{jk}} + C_{O_{jk}} + C_{M_{jk}} + C_{L_{jk}} \tag{1}$$

where $C_{jk} > 0$. The deployment-cost matrix $\boldsymbol{C}$ of deploying L cloudlets across M candidate points and can be expressed as follows:

$$\boldsymbol{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1M} \\ C_{21} & C_{22} & \dots & C_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ C_{L1} & C_{L2} & \dots & C_{LM} \end{bmatrix} \tag{2}$$

3.2. Latency Model

In contrast to distance-only formulations [7], we introduce a comprehensive latency model that decomposes end-to-end service time into propagation, transmission, processing, and queuing delays, each parameterized by realistic network characteristics.

- **Propagation Delay**: propagation delay $t_{\mathrm{P}_{ik}}$ constitutes the time required for a signal to travel from an end device e_i to a candidate point p_k and given as.

$$t_{\mathrm{P}_{ik}} = \frac{d_{ik}}{v_s} \quad \text{(seconds)} \tag{3}$$

  where v_s denotes the signal propagation speed in the communication medium and d_{ik} denotes the distance between the end device e_i and the candidate point p_k.

- **Transmission delay**: The transmission delay t_{T_i} accounts for bandwidth limitations and data serialization requirements, reflecting the time required to transmit a task of size τ_{e_i} over the available network bandwidth B. It can be given as

$$t_{\mathrm{T}_i} = \frac{\tau_{e_i}}{B} \quad \text{(seconds)} \tag{4}$$

Table 2. Main notation used in the system model and LCRA framework.

Notation	Description
$\mathcal{P}$	Set of candidate points.
p_k	Candidate point k, representing a feasible cloudlet deployment location.
$\mathcal{P}_{uo}$	Set of currently unoccupied candidate points.
$\mathcal{C}$	Set of heterogeneous cloudlets.
c_j	Cloudlet j in the cloudlet set $\mathcal{C}$.
$\mathcal{R}_{c_j}$	Resource specification of cloudlet c_j.
$\{\mathcal{R}_{c_j}\}$	Set of cloudlet capacity profiles for all cloudlets.
$P_{c_j}, m_{c_j}, s_{c_j}$	Processor, memory, and storage capacity of cloudlet c_j.
$\mathcal{U}_{c_j}$	Current processor, memory, and storage usage of cloudlet c_j.
$\widetilde{\mathcal{U}}_{jk}$	Projected processor, memory, and storage usage of cloudlet c_j if it is deployed at candidate point p_k.
$\{\mathcal{U}_{c_j}\}$	Updated usage vectors of the deployed cloudlets.
r_{c_j}	Communication radius of cloudlet c_j.
$\{r_{c_j}\}$	Set of communication radii for all cloudlets.
τ_{c_j}	Categorical cloudlet type, such as small, medium, or large.
$\mathcal{C}_{ud}$	Set of cloudlets that have not yet been deployed in a placement vector.
$\mathcal{E}$	Set of end devices.
e_i	End device i responsible for generating computational tasks.
$\mathcal{R}_{e_i}$	Resource demand vector of end device e_i.
$\{\mathcal{R}_{e_i}\}$	Set of device demand vectors for all end devices.
$P_{e_i}, m_{e_i}, s_{e_i}$	Processor, memory, and storage demand of end device e_i.
τ_{e_i}	Task size generated by end device e_i.
$\mathcal{E}_{uc}$	Set of uncovered end devices.
$\mathcal{E}_{jk}^{cov}$	Set of end devices covered by cloudlet c_j when deployed at candidate point p_k.
$\mathcal{E}_{jk}^{srv}$	Set of serviceable end devices, defined as the subset of $\mathcal{E}_{jk}^{cov}$ that can be admitted by cloudlet c_j without exceeding its processor, memory, or storage capacity.
C_{jk}	Cost of deploying cloudlet c_j at candidate point p_k, expressed in V_u.
$\boldsymbol{C}$	Deployment-cost matrix whose entries are expressed in V_u.
V_u	Virtual, user-defined unit for deployment-cost quantities, with no fixed conversion to real-world currency.
d_{ik}	Distance between end device e_i and candidate point p_k.
$\mathcal{D}$	Distance set containing end-device–candidate-point distances.
ℓ_{ijk}	Total latency when end device e_i is served by cloudlet c_j deployed at candidate point p_k.
$\boldsymbol{t}_{\mathrm{S}}$	End-device–candidate-point latency structure containing the static communication-latency values.
$t_{\mathrm{S}}^{\max}$	Maximum static communication latency in the considered problem instance.
$\mathbf{L}$	Dynamic block latency structure containing the latency values for all end-device–cloudlet–candidate-point associations.
$\mathbf{L}_i$	$M \times L$ latency block associated with end device e_i, whose rows correspond to candidate points and columns correspond to cloudlets.
Ω	Assignment set produced by PBDA; its binary form is represented by w_{ijk}, where $w_{ijk} = 1$ when end device e_i is assigned to cloudlet c_j at candidate point p_k.
τ_{up}	Upgrade threshold used to flag highly utilized deployed cloudlets.
Q_{up}	Set of deployed cloudlet–candidate-point pairs flagged by PBDA or CRR for possible resource upgrading and processed by CRR.

- **Processing Delay**: The processing delay $t_{\mathrm{C}_{ijk}}$ is the time required to execute a task of size τ_{e_i} from end device e_i on cloudlet c_j deployed at candidate point p_k. It depends on the task size, instruction density D, and cloudlet processing capacity P_{c_j}. The instruction density converts the task size into the required CPU cycles. Thus, the processing delay is given by

$$t_{\mathrm{C}_{ijk}} = \frac{D\,\tau_{e_i}}{P_{c_j} \times 10^9} \quad \text{(seconds)}. \tag{5}$$

- **Queuing Delay**: To capture resource contention at deployed cloudlets, the latency model includes a finite-source $M/M/1/N_k$ queuing component [43], [44]. The $M/M/1/N_k$ model describes a single-server queue with *Poisson* task generation by each idle source and *exponentially distributed* service times, while explicitly incorporating a finite population of potential task sources. In this context, N_k denotes the finite number of end devices served by the cloudlet deployed at candidate point p_k. When cloudlet c_j is deployed at point p_k, the queue at c_j serves N_k devices. The finite-source assumption represents a bounded population of assigned devices: devices with outstanding tasks do not generate another task until their current task is completed.

  Let X_j be a random variable representing the number of tasks at cloudlet c_j, with probability distribution $x_{j_n} = \Pr[X_j = n]$, where $n = 0, 1, \ldots, N_k$. Let λ_j denote the task-generation rate of each idle end device served by cloudlet c_j, and let μ_j denote the service rate of cloudlet c_j. For the finite-source queuing model, all end devices are assumed to generate tasks of the same size, such that $\tau_{e_i} = \tau$ for every end device e_i, consistent with the fixed task-size configuration used in the experiments.

$$\mu_j = \frac{P_{c_j} \times 10^9}{D \times \tau} \quad \text{(tasks/second)} \tag{6}$$

  When the system contains n tasks, only $N_k - n$ end devices are idle and capable of generating a new task. Therefore, the state-dependent total arrival rate and service rate are defined as

$$\lambda_{j,n} = (N_k - n)\lambda_j, \qquad n = 0, 1, \ldots, N_k - 1,$$

  and

$$\mu_{j,n} = \mu_j, \qquad n = 1, 2, \ldots, N_k,$$

  respectively. The traffic intensity is defined as $\rho_j = \lambda_j/\mu_j$. The steady-state probabilities x_{j_n} for the finite-source $M/M/1/N_k$ queuing system are given in [43] , [44] as:

$$\begin{aligned} x_{j_n} &= \frac{N_k!}{(N_k - n)!}\rho_j^n x_{j_0}, \\ \text{where} \qquad x_{j_0} &= \left[\sum_{n=0}^{N_k} \frac{N_k!}{(N_k - n)!}\rho_j^n\right]^{-1} \end{aligned} \tag{7}$$

  The expected number of tasks, denoted by $E[X_j]$, in the system at cloudlet c_j is expressed as:

$$E[X_j] = \sum_{n=1}^{N_k} n\, x_{j_n}, \tag{8}$$

  The effective arrival rate, denoted by $\lambda_{\text{eff},j,k}$, accounts for the finite number of task sources and is expressed as:

$$\lambda_{\text{eff},j,k} = \sum_{n=0}^{N_k - 1} \lambda_{j,n} x_{j_n} = (N_k - E[X_j])\, \lambda_j, \tag{9}$$

  where $(N_k - E[X_j])$ represents the expected number of idle end devices capable of generating new tasks for cloudlet c_j. Using Little's Law [45], the average response time is defined as:

$$R_{j,k} = \frac{E[X_j]}{\lambda_{\text{eff},j,k}},$$

  and the corresponding queuing delay is given by:

$$t_{\text{Q}_{jk}} = \max\left(0,\, R_{j,k} - \frac{1}{\mu_j}\right) \quad \text{(seconds)}. \tag{10}$$

If $N_k = 0$, no end device is assigned to the cloudlet at candidate point p_k, and its queuing delay is defined as zero. The queueing model provides a consistent analytical basis for comparing placement methods under the stated workload assumptions. The reported latency results should be interpreted within this setting; validation against measured edge workloads was not undertaken. Bursty arrivals, more general service-time distributions, or parallel processing would require an appropriately adapted queueing model.

With reference to equations (3)-(10), the total latency ℓ_{ijk} for end device e_i served by cloudlet c_j deployed at candidate point p_k is given as:

$$\ell_{ijk} = t_{\mathrm{T}_i} + t_{\mathrm{P}_{ik}} + t_{\mathrm{C}_{ijk}} + t_{\mathrm{Q}_{jk}}, \tag{11}$$

To organize the latency values associated with all end-device–cloudlet–candidate-point combinations, the latency structure is represented by the block matrix

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \\ \vdots \\ \mathbf{L}_N \end{bmatrix}, \qquad \mathbf{L}_i \in \mathbb{R}_{\geq 0}^{M \times L},$$

where $\mathbf{L}_i$ is the latency values associated with end device e_i and can be written as

$$\mathbf{L}_i = \begin{bmatrix} \ell_{i11} & \ell_{i21} & \cdots & \ell_{iL1} \\ \ell_{i12} & \ell_{i22} & \cdots & \ell_{iL2} \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{i1M} & \ell_{i2M} & \cdots & \ell_{iLM} \end{bmatrix}.$$

The complete latency structure $\mathbf{L}$ is not treated as a permanently fixed matrix. The processing component $t_{\mathrm{C}_{ijk}}$ depends on the considered cloudlet, while the queueing component $t_{\mathrm{Q}_{jk}}$ depends on the current assignment and queue state. Therefore, the required entries of $\mathbf{L}$ are evaluated on demand, or $\mathbf{L}$ may be temporarily constructed as a snapshot and updated whenever the assignment or queue state changes. The communication-latency component associated with end device e_i and candidate point p_k is defined as

$$t_{S_{ik}} = t_{T_i} + t_{P_{ik}}, \tag{12}$$

Since these components depend only on the end device and the candidate point, they can be computed and stored in the reusable communication-latency structure

$$\mathbf{t}_S = \{t_{S_{ik}} \mid i = 1, \ldots, N;\ k = 1, \ldots, M\}. \tag{13}$$

4. Problem Formulation

The CPP jointly determines where heterogeneous cloudlets should be deployed and how end devices should be assigned to them. The objective is to minimize both deployment cost and service latency while satisfying coverage, assignment, resource-capacity, and deployment-feasibility requirements.

Decision Variables. Four binary decision variables describe the placement and assignment decisions. The deployment variable y_{jk} indicates whether cloudlet c_j is deployed at candidate point p_k:

$$y_{jk} = \begin{cases} 1, & \text{if cloudlet } c_j \text{ is deployed at } p_k, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

The assignment variable x_{ij} indicates whether end device e_i is assigned to cloudlet c_j:

$$x_{ij} = \begin{cases} 1, & \text{if end device } e_i \text{ is assigned to } c_j, \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

The service-association variable w_{ijk} indicates whether end device e_i is served by cloudlet c_j deployed at candidate point p_k:

$$w_{ijk} = \begin{cases} 1, & \text{if } e_i \text{ is served by } c_j \text{ deployed at } p_k, \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

Finally, the service-location variable z_{ik} indicates whether end device e_i is served from candidate point p_k:

$$z_{ik} = \begin{cases} 1, & \text{if } e_i \text{ is served from } p_k, \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

Objective Functions. The first objective minimizes the total deployment cost:

$$f_C = \sum_{j=1}^{L} \sum_{k=1}^{M} C_{jk} y_{jk}, \tag{18}$$

Since C_{jk} is expressed in the virtual cost unit V_u, f_C is expressed in the same unit. The second objective minimizes the total service latency:

$$f_L = \sum_{i=1}^{N} \sum_{j=1}^{L} \sum_{k=1}^{M} \ell_{ijk} w_{ijk}, \tag{19}$$

Bi-objective CPP Formulation. Using the above decision variables and objectives, the CPP is formulated as

$$\mathbf{P}: \quad \underset{\{y_{jk},\, x_{ij},\, w_{ijk},\, z_{ik}\}}{\text{minimize}} \quad (f_C, f_L) \tag{20a}$$

Coverage and assignment constraints:

$$\sum_{j=1}^{L} \sum_{k=1}^{M} w_{ijk} = 1, \qquad \forall i = 1, \ldots, N, \tag{20b}$$

$$w_{ijk} \le \min\left(y_{jk}, \mathbb{1}[d_{ik} \le r_{c_j}]\right), \qquad \forall i, j, k, \tag{20c}$$

$$\sum_{k=1}^{M} w_{ijk} = x_{ij}, \qquad \forall i, j, \tag{20d}$$

$$\sum_{j=1}^{L} w_{ijk} = z_{ik}, \qquad \forall i, k, \tag{20e}$$

$$z_{ik} \le \sum_{j=1}^{L} y_{jk}, \qquad \forall i, k, \tag{20f}$$

$$\sum_{j=1}^{L} x_{ij} \le 1, \qquad \forall i, \tag{20g}$$

$$\sum_{k=1}^{M} z_{ik} = 1, \qquad \forall i. \tag{20h}$$

Resource-capacity constraints:

$$\sum_{i=1}^{N}\sum_{k=1}^{M} w_{ijk} m_{e_i} \le m_{c_j} \sum_{k=1}^{M} y_{jk}, \qquad \forall j, \tag{20i}$$

$$\sum_{i=1}^{N}\sum_{k=1}^{M} w_{ijk} s_{e_i} \le s_{c_j} \sum_{k=1}^{M} y_{jk}, \qquad \forall j, \tag{20j}$$

$$\sum_{i=1}^{N}\sum_{k=1}^{M} w_{ijk} P_{e_i} \le P_{c_j} \sum_{k=1}^{M} y_{jk}, \qquad \forall j. \tag{20k}$$

Deployment-feasibility constraints:

$$\sum_{j=1}^{L} y_{jk} \le 1, \qquad \forall k, \tag{20l}$$

$$\sum_{k=1}^{M} y_{jk} \le 1, \qquad \forall j, \tag{20m}$$

$$x_{ij} \le \sum_{k=1}^{M} y_{jk}, \qquad \forall i, j. \tag{20n}$$

Constraint Interpretation. Constraint (20b) requires every end device to be served by exactly one cloudlet–candidate-point pair. Constraint (20c) permits a service association only when the corresponding cloudlet is deployed and the end device lies within its communication radius. Constraints (20d) and (20e) maintain consistency between the detailed service association w_{ijk} and the cloudlet- and location-level assignment variables x_{ij} and z_{ik}, respectively. Constraints (20f)–(20h) enforce valid and unique service locations.

Constraints (20i)–(20k) ensure that the total memory, storage, and processor demands assigned to each cloudlet do not exceed its corresponding capacities. Finally, Constraints (20l) and (20m) enforce the one-cloudlet-per-point and one-location-per-cloudlet deployment structure, while Constraint (20n) prevents assignment to an undeployed cloudlet.

Normalization of the Objectives. Because deployment cost and service latency have different units and numerical scales, they are normalized before being combined through weighted-sum scalarization, a widely used method for incorporating decision-maker preferences [46, 47]:

$$\bar{f}_C = \frac{f_C}{f_C^{\max}}, \tag{21}$$

$$\bar{f}_L = \frac{f_L}{f_L^{\max}}. \tag{22}$$

The implementation uses the following fixed deployment-cost normalization reference, accounting for potential distance-based radius-extension costs:

$$f_C^{\max} = \sum_{j=1}^{L} \max_{k\in\{1,\dots,M\}} \left[C_{jk} \max\left(1, \frac{\max_{i\in\{1,\dots,N\}} d_{ik}}{r_{c_j}}\right)\right], \tag{23}$$

This reference is calculated once from the instance's nominal costs, nominal radii, and device-to-candidate-point distances. It does not charge an extension cost to every solution: additional cost is included in a candidate's objective only when an extension actually occurs, according to Subsection 5.3. The reference is not an optimal deployment cost or an optimality certificate.

The maximum latency considered for end device e_i is

$$\ell_i^{\max} = \max_{\substack{j=1,\dots,L \\ k=1,\dots,M \\ N_k \in \{1,\dots,N\}}} \left(t_{\mathrm{S}_{ik}} + t_{\mathrm{C}_{ijk}} + t_{\mathrm{Q}_{jk}} \right), \tag{24}$$

and hence

$$f_L^{\max} = \sum_{i=1}^{N} \ell_i^{\max}. \tag{25}$$

These normalization references are determined from each problem instance before optimization and remain fixed throughout the search. Consequently, the normalized objectives are dimensionless quantities evaluated on a common scale.

Preference-Dependent Weighting. The relative importance of deployment cost and latency is controlled by two nonnegative weights, denoted by ω_C and ω_L, respectively:

$$\omega_C \geq 0, \qquad \omega_L \geq 0, \qquad \omega_C + \omega_L = 1. \tag{26}$$

The weights are determined by the preference parameter $\alpha \in [0, 1]$ as

$$(\omega_C(\alpha), \omega_L(\alpha)) = \begin{cases} \left(\dfrac{\alpha^2}{\alpha^2 + (1-\alpha)^{0.5}}, \dfrac{(1-\alpha)^{0.5}}{\alpha^2 + (1-\alpha)^{0.5}} \right), & 0 \leq \alpha \leq 0.3, \\ (\alpha, 1-\alpha), & 0.3 < \alpha < 0.8, \\ \left(\dfrac{\alpha^{0.5}}{\alpha^{0.5} + (1-\alpha)^2}, \dfrac{(1-\alpha)^2}{\alpha^{0.5} + (1-\alpha)^2} \right), & 0.8 \leq \alpha \leq 1. \end{cases} \tag{27}$$

The preference-dependent weighting policy provides three regimes for expressing the relative importance of latency and deployment cost:

Latency-focused regime ($0 \leq \alpha \leq 0.3$)**:** The nonlinear weighting rule gives latency substantially greater importance than deployment cost, reflecting a preference for low-latency solutions.

Balanced trade-off regime ($0.3 < \alpha < 0.8$)**:** The cost weight equals α, and the latency weight equals $1 - \alpha$. Increasing α gradually shifts importance from latency toward deployment cost. "Balanced" therefore means that both objectives participate in the trade-off, not that their weights are always equal. Equal weighting occurs at $\alpha = 0.5$.

Cost-focused regime ($0.8 \leq \alpha \leq 1$)**:** The nonlinear weighting rule gives deployment cost substantially greater importance than latency, reflecting a preference for lower-cost solutions.

In the latency-focused regime, the formula strengthens the latency weight by taking the square root of its preference term and reduces the cost weight by squaring its term. In the cost-focused regime, these operations are reversed to strengthen the cost weight. The resulting terms are divided by their sum so that the two weights add up to one. The powers 2 and 0.5 are fixed design choices, not values selected through experimental optimization.

Switching between the linear and nonlinear rules causes jumps in the weights at $\alpha = 0.3$ and $\alpha = 0.8$. These thresholds are fixed design settings. Figure 2 illustrates the mathematical weighting policy, not experimentally measured performance improvements.

The experiments evaluate $\alpha = 0.1, 0.5$, and 0.9, representing the three regimes. Performance immediately around the boundaries has not been experimentally evaluated.

Final Scalar Optimization Problem. For a fixed value of α, the normalized objectives are combined into the scalar objective

$$F_\alpha := \omega_C(\alpha) \bar{f}_C + \omega_L(\alpha) \bar{f}_L. \tag{28}$$

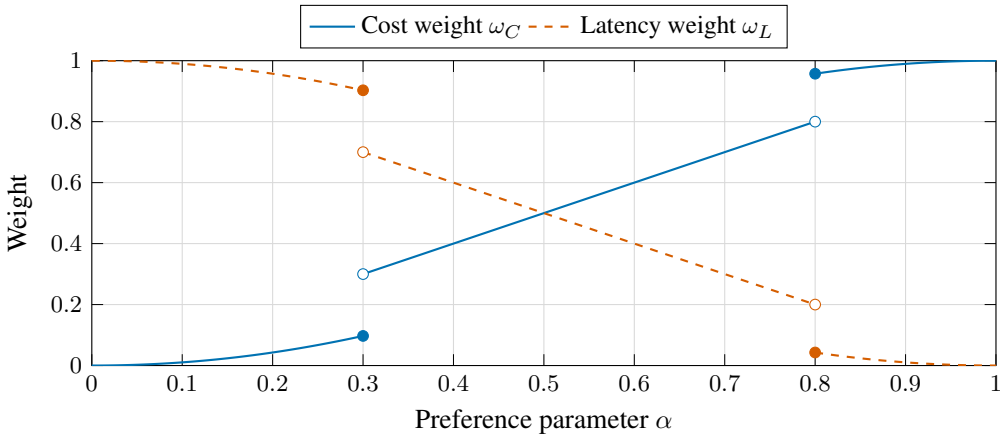


Figure 2. Analytical cost and latency weights as functions of the preference parameter. Open and filled markers distinguish excluded and included branch endpoints, respectively.

Substituting Equations (18) and (19) gives

$$F_\alpha := \omega_C(\alpha)\frac{\sum_{j=1}^{L}\sum_{k=1}^{M} C_{jk}y_{jk}}{f_C^{\max}} + \omega_L(\alpha)\frac{\sum_{i=1}^{N}\sum_{j=1}^{L}\sum_{k=1}^{M} \ell_{ijk}w_{ijk}}{f_L^{\max}}. \tag{29}$$

Accordingly, for a fixed preference parameter $\alpha \in [0,1]$, the scalar CPP is formulated as

$$\mathbf{P}_\alpha : \quad \underset{\{y_{jk},\, x_{ij},\, w_{ijk},\, z_{ik}\}}{\text{minimize}} \; F_\alpha, \tag{30}$$

subject to Constraints (20b)–(20n). The above formulation describes the nominal cloudlet configuration. The implementation additionally permits the radius-extension fallback defined in Subsection 5.3. When this fallback is applied, candidate coverage, deployment cost, service-link penalties, and scalar fitness are evaluated using the effective parameters specified in that subsection.

5. Latency-Cost-Requirement-Aware Cloudlet Placement Framework (LCRA)

The LCRA framework generates and refines informed cloudlet-placement candidates independently of the metaheuristic update operator. By replacing purely random initialization with solutions that reflect key CPP objectives, LCRA provides a common placement representation, assignment procedure, repair mechanism, and fitness function across different metaheuristics, enabling a fair comparison focused on their search behavior

5.1. Cloudlet Placement Initialization

This subsection defines the placement-vector representation used by the initialization strategies. Let

$$\mathbf{\Psi} = (\psi_1, \psi_2, \ldots, \psi_k, \ldots, \psi_M)$$

denote a placement vector where ψ_k is the index of the cloudlet placed at candidate point p_k. If no cloudlet is assigned to candidate point p_k, then $\psi_k = 0$; otherwise, $\psi_k = j$ indicates that cloudlet c_j is deployed at p_k, where $j \in \{1, 2, \ldots, L\}$. For example, for $M = 5$ candidate points and $L = 3$ cloudlets, the placement vector $\mathbf{\Psi} = (3, 0, 1, 0, 2)$ indicates that cloudlet c_3 is deployed at p_1, no cloudlet is deployed at p_2, cloudlet c_1 is deployed at p_3, no cloudlet is deployed at p_4, and cloudlet c_2 is deployed at p_5. The initialization strategy is selected according to

the optimization priority defined by the preference parameter α. The placement vector specifies cloudlet locations. The complete implementation state additionally contains candidate-specific effective radii, extension costs, and the associated assignment state, as described in Subsection 5.3. Assignment and fitness evaluation use this complete state.

5.1.1. Latency-Oriented Initialization The Latency-Aware Coverage Initialization (LACI) algorithm constructs the coverage-oriented component in every preference regime. This component receives its largest quota under latency-dominant preferences, $\alpha \leq 0.3$. LACI constructs the placement vector sequentially by assigning cloudlets to feasible candidate points that favor low-latency service. The algorithm can be expressed as the following sequence of operations:

1. Initialize the set of uncovered end devices $\mathcal{E}_{uc} := \mathcal{E}$, the set of undeployed cloudlets $\mathcal{C}_{ud} := \mathcal{C}$, the set of unoccupied candidate points $\mathcal{P}_{uo} := \mathcal{P}$ and the placement vector $\Psi := \mathbf{0}_{1\times M}$.

2. For each cloudlet c_j, the minimum deployment cost over all candidate points is identified. The maximum allowable deployment budget is then computed by summing these minimum costs and scaling the result by the budget factor B_f:
$$B_{\max} = B_f \times \sum_{j=1}^{L} \min_{k\in\{1,\ldots,M\}} C_{jk}. \tag{31}$$

3. For each cloudlet $c_j \in \mathcal{C}_{ud}$, initialize its feasible candidate-point index set and score collection as $\mathcal{F}_j := \emptyset$ and $\mathcal{G}_j := \emptyset$ respectively, where $\mathcal{G}_j$ stores the candidate-point scores associated with cloudlet c_j.

4. For each candidate point $p_k \in \mathcal{P}_{uo}$, construct the set of currently uncovered end devices that lie within the communication radius of cloudlet c_j as $\mathcal{E}_{jk}^{\text{cov}} := \{e_i \in \mathcal{E}_{uc} \mid d_{ik} \leq r_{c_j}\}$.

5. For each pair (c_j, p_k), construct the serviceable-device set $\mathcal{E}_{jk}^{srv}$ by identifying the covered end devices that can be admitted without exceeding the resource capacities of cloudlet c_j. The set is initialized as $\mathcal{E}_{jk}^{srv} := \emptyset$, while the projected resource-usage vector is initialized as $\widetilde{\mathcal{U}}_{jk} := [0, 0, 0]$. The covered devices in $\mathcal{E}_{jk}^{cov}$ are then processed in ascending order of their static communication latency $t_{\mathrm{S}_{ik}}$, with ties resolved by the device index i. An end device e_i is admitted only if $\widetilde{\mathcal{U}}_{jk}[r] + \mathcal{R}_{e_i}[r] \leq \mathcal{R}_{c_j}[r]$ for every resource dimension $r \in \{1, 2, 3\}$. When this condition is satisfied, e_i is added to $\mathcal{E}_{jk}^{srv}$, and the projected usage is updated as $\widetilde{\mathcal{U}}_{jk} := \widetilde{\mathcal{U}}_{jk} + \mathcal{R}_{e_i}$. This procedure ensures that only latency-prioritized, resource-feasible covered devices are included in the serviceable-device set for the evaluated placement.

6. Candidate point p_k is considered feasible for cloudlet c_j only if $B_c + C_{jk} \leq B_{\max}$ and $\mathcal{E}_{jk}^{\text{srv}} \neq \emptyset$.

7. For each feasible pair (c_j, p_k), compute the coverage-quality π_{jk}^{CQ}, static-latency π_{jk}^{LT}, load-balancing π_{jk}^{LB}, and resource-efficiency π_{jk}^{RE} scores. The aggregate placement score is then given by
$$g_{jk} := \gamma_1\pi_{jk}^{CQ} + \gamma_2\pi_{jk}^{LT} + \gamma_3\pi_{jk}^{LB} + \gamma_4\pi_{jk}^{RE}. \tag{32}$$

8. Add the candidate-point index to the feasible set and store its corresponding aggregate score:
$$\mathcal{F}_j := \mathcal{F}_j \cup \{k\}, \qquad \mathcal{G}_j := \mathcal{G}_j \cup \{(k, g_{jk})\}.$$
Thus, $\mathcal{G}_j$ maintains the score associated with every feasible candidate point considered for cloudlet c_j.

9. After all $p_k \in \mathcal{P}_{uo}$ have been evaluated, the stored scores in $\mathcal{G}_j$ are used to select the feasible candidate point with the largest aggregate score:
$$\nu_j := \min\left(\arg\max_{k\in\mathcal{F}_j} g_{jk}\right), \qquad \mathcal{F}_j \neq \emptyset. \tag{33}$$
The outer $\min(\cdot)$ resolves score ties by selecting the smallest candidate-point index.

10. Deploy cloudlet c_j at candidate point p_{ν_j} by setting $\Psi[\nu_j] := j$.Remove c_j from $\mathcal{C}_{ud}$ and p_{ν_j} from $\mathcal{P}_{uo}$, and update the accumulated deployment cost $B_c := B_c + C_{j\nu_j}$.

11. Commit the expected resource usage of the selected placement $\mathcal{U}_{c_j} := \widetilde{\mathcal{U}}_{j\nu_j}$

12. Repeat the evaluation and selection procedure for the remaining undeployed cloudlets until no additional feasible placement can be made or all cloudlets have been processed.

Coverage Quality Score π_{jk}^{CQ} The coverage quality score π_{jk}^{CQ} evaluates the suitability of deploying cloudlet c_j at candidate point p_k in terms of the quality of coverage provided to the serviceable end devices. It is computed as follows. For each serviceable end device $e_i \in \mathcal{E}_{jk}^{\text{srv}}$, the normalized coverage-proximity ratio relative to candidate point p_k is defined as $\beta_{ijk} = 1 - \frac{d_{ik}}{r_{c_j}}$, where larger values indicate that e_i is closer to p_k relative to the communication radius of cloudlet c_j. To give greater preference to devices located closer to the candidate point, a proximity-quality weight is assigned as

$$q_{ijk} = \begin{cases} 1.0, & \beta_{ijk} > 0.8, \\ 0.8, & 0.6 < \beta_{ijk} \leq 0.8, \\ 0.6, & 0.4 < \beta_{ijk} \leq 0.6, \\ 0.4, & \text{otherwise.} \end{cases}$$

The contribution of each serviceable device to the coverage quality of placement (c_j, p_k) is therefore $\beta_{ijk} \times q_{ijk}$. The overall normalized coverage-quality score is computed as

$$\pi_{jk}^{\text{CQ}} = \frac{1}{|\mathcal{E}_{uc}|} \sum_{e_i \in \mathcal{E}_{jk}^{\text{srv}}} \beta_{ijk} q_{ijk}, \qquad |\mathcal{E}_{uc}| > 0. \tag{34}$$

Static Communication Latency Score π_{jk}^{LT} The LACI algorithm evaluates candidate placements using the static communication latency $t_{\text{S}_{ik}}$, since the processing and queueing components of the complete latency are assignment-dependent and are not yet available during initialization. For each problem instance, the maximum static communication latency is first determined as

$$t_{\text{S}}^{\max} = \max_{\substack{i=1,\ldots,N \\ k=1,\ldots,M}} t_{\text{S}_{ik}}, \tag{35}$$

and used as the normalization reference. The device-level latency score is then defined as

$$s_{ik}^{\text{LT}} = \begin{cases} 1, & t_{\text{S}}^{\max} = 0, \\ 1 - \dfrac{t_{\text{S}_{ik}}}{t_{\text{S}}^{\max}}, & t_{\text{S}}^{\max} > 0. \end{cases} \tag{36}$$

This formulation assigns larger scores to lower communication-latency values. The placement-level latency score is obtained by averaging the device-level scores over the serviceable-device set:

$$\pi_{jk}^{\text{LT}} = \frac{1}{|\mathcal{E}_{jk}^{\text{srv}}|} \sum_{e_i \in \mathcal{E}_{jk}^{\text{srv}}} s_{ik}^{\text{LT}}, \qquad |\mathcal{E}_{jk}^{\text{srv}}| > 0. \tag{37}$$

Consequently, a larger π_{jk}^{LT} indicates lower average static communication latency for the devices that can be feasibly served by the evaluated cloudlet–candidate-point pair (c_j, p_k).

Load Balancing Score π_{jk}^{LB} Let $\boldsymbol{\eta}$ denote the normalized workloads of the cloudlets already selected in the current placement. For an evaluated cloudlet–candidate-point pair (c_j, p_k), the projected workload of cloudlet c_j is first computed from its utilization across the three considered resource dimensions. Specifically,

$$\widetilde{\eta}_{jk} = \frac{1}{3}\sum_{r=1}^{3}\frac{\widetilde{\mathcal{U}}_{jk}[r]}{\mathcal{R}_{c_j}[r]}, \tag{38}$$

where $\widetilde{\mathcal{U}}_{jk}[r]$ is the projected usage of resource r, and $\mathcal{R}_{c_j}[r]$ is the corresponding capacity of cloudlet c_j. Thus, $\widetilde{\eta}_{jk}$ represents the average normalized resource utilization that would result if c_j were deployed at p_k.

To evaluate the effect of this candidate placement on workload distribution, $\widetilde{\eta}_{jk}$ is appended to the workloads of the already selected cloudlets:

$$\boldsymbol{\eta}_{jk} = (\boldsymbol{\eta}, \widetilde{\eta}_{jk})\,. \tag{39}$$

The workload imbalance introduced by the candidate placement is measured by the difference between the maximum and minimum normalized workloads in $\boldsymbol{\eta}_{jk}$. Accordingly, the load-balancing score is defined as

$$\pi_{jk}^{\mathrm{LB}} = 1 - [\max(\boldsymbol{\eta}_{jk}) - \min(\boldsymbol{\eta}_{jk})]\,. \tag{40}$$

Since the normalized workloads lie in $[0, 1]$, the score π_{jk}^{LB} also lies in $[0, 1]$. A larger value indicates a smaller workload spread and, consequently, a more balanced distribution among the selected cloudlets. For the first selected cloudlet, $\boldsymbol{\eta}_{jk}$ contains only one workload value; therefore, its maximum and minimum coincide and $\pi_{jk}^{\mathrm{LB}} = 1$.

Resource Efficiency Score π_{jk}^{RE} For each resource dimension $r \in \{1, 2, 3\}$, the projected normalized utilization of cloudlet c_j at candidate point p_k is defined as

$$u_{jk,r} = \frac{\widetilde{\mathcal{U}}_{jk}[r]}{\mathcal{R}_{c_j}[r]}, \qquad 0 \le u_{jk,r} \le 1, \tag{41}$$

where $\widetilde{\mathcal{U}}_{jk}[r]$ denotes the projected usage of resource r, and $\mathcal{R}_{c_j}[r]$ denotes the corresponding capacity of cloudlet c_j. The resource-efficiency score evaluates how closely the projected utilization of the three resource dimensions approaches a predefined target utilization $u_{\mathrm{tar}} \in [0, 1]$. It is defined as

$$\pi_{jk}^{\mathrm{RE}} = \begin{cases} 0, & \widetilde{\mathcal{U}}_{jk} = \mathbf{0}, \\ \dfrac{1}{3}\displaystyle\sum_{r=1}^{3}\left(1 - |u_{\mathrm{tar}} - u_{jk,r}|\right), & \text{otherwise.} \end{cases} \tag{42}$$

A resource utilization equal to u_{tar} yields the maximum contribution for that resource, whereas increasing deviation from the target reduces its contribution. Consequently, a larger π_{jk}^{RE} indicates that the projected processor, memory, and storage utilizations are collectively closer to the desired operating level. A score of zero is assigned when no resources are utilized, preventing an empty placement from receiving a positive resource-efficiency value.

Since the coverage-quality, static-latency, load-balancing, and resource-efficiency indicators are all normalized to $[0, 1]$, and the weights satisfy $\gamma_h \ge 0$ and $\sum_{h=1}^{4}\gamma_h = 1$, the aggregate placement score also satisfies $g_{jk} \in [0, 1]$. After selecting candidate point p_{ν_j}, LACI commits the corresponding projected resource usage, updates the workload collection and accumulated deployment cost, removes the admitted devices from $\mathcal{E}_{uc}$, and removes c_j and p_{ν_j} from $\mathcal{C}_{ud}$ and $\mathcal{P}_{uo}$, respectively. If $\mathcal{F}_j = \emptyset$, cloudlet c_j remains undeployed. The serviceable-device sets constructed during LACI are used only to guide initialization; the final end-device assignments are subsequently reconstructed by PBDA.

Algorithm 1: Latency-Aware Coverage Initialization (LACI)

Input: $\mathcal{E}$, $\mathcal{C}$, ordered cloudlet sequence $\boldsymbol{c}_{ord}$, $\mathcal{P}$, $\boldsymbol{C}$, $\mathcal{D}$, $\{r_{c_j}\}$, $\{\mathcal{R}_{e_i}\}$, $\{\mathcal{R}_{c_j}\}$, $\boldsymbol{t}_{\mathrm{S}}$, $\gamma_1, \ldots, \gamma_4$, and B_f.
Output: Placement vector $\boldsymbol{\Psi}$.
// Initialize the placement and state variables:
Initialize $\mathcal{E}_{uc} := \mathcal{E}$, $\mathcal{C}_{ud} := \mathcal{C}$, and $\mathcal{P}_{uo} := \mathcal{P}$, $\boldsymbol{\Psi} := \mathbf{0}_{1\times M}$, $B_c := 0$, and $\boldsymbol{\eta} := \emptyset$
Initialize $\mathcal{U}_{c_j} := [0, 0, 0]$ for every $c_j \in \mathcal{C}$
Compute $t_{\mathrm{S}}^{\max}$ using Equation (35)
// Compute the deployment-budget limit:
Compute $totalCost := \sum_{j=1}^{L} \min_k C_{jk}$ and $B_{\max} := B_f\ totalCost$
// Evaluate each cloudlet at the available candidate points:
foreach $c_j \in \boldsymbol{c}_{ord}$ **do**
  Initialize $\mathcal{F}_j := \emptyset$
  foreach $p_k \in \mathcal{P}_{uo}$ **do**
    if $B_c + C_{jk} > B_{\max}$ **then**
      continue
    // Construct the covered and serviceable-device sets and projected usage:
    Construct $\mathcal{E}_{jk}^{\mathrm{cov}} = \{e_i \in \mathcal{E}_{uc} \mid d_{ik} \le r_{c_j}\}$
    Construct $\mathcal{E}_{jk}^{\mathrm{srv}}$ and $\widetilde{\mathcal{U}}_{jk}$ by admitting devices in ascending $t_{\mathrm{S}_{ik}}$ order, subject to the resource capacities, and cumulatively updating $\widetilde{\mathcal{U}}_{jk}$ with each admitted device's demand
    if $\mathcal{E}_{jk}^{\mathrm{srv}} = \emptyset$ **then**
      continue
    // Evaluate the feasible candidate placement:
    Compute π_{jk}^{CQ} using Equation (34)
    Compute π_{jk}^{LT} using Equation (37)
    Compute π_{jk}^{LB} using Equation (40)
    Compute π_{jk}^{RE} using Equation (42)
    Compute g_{jk} using Equation (32)
    Set $\mathcal{F}_j := \mathcal{F}_j \cup \{k\}$
  // Select a candidate point and update the placement state:
  if $\mathcal{F}_j \neq \emptyset$ **then**
    Select ν_j using Equation (33)
    Set $\psi_{\nu_j} := j$
    Set $\mathcal{U}_{c_j} := \widetilde{\mathcal{U}}_{j\nu_j}$
    Append the normalized workload of the selected placement to $\boldsymbol{\eta}$
    Set $B_c := B_c + C_{j\nu_j}$
    Set $\mathcal{E}_{uc} := \mathcal{E}_{uc} \setminus \mathcal{E}_{j\nu_j}^{\mathrm{srv}}$
    Set $\mathcal{C}_{ud} := \mathcal{C}_{ud} \setminus \{c_j\}$
    Set $\mathcal{P}_{uo} := \mathcal{P}_{uo} \setminus \{p_{\nu_j}\}$

5.1.2. Balanced Cost–Latency Initialization For balanced cost–latency preferences, the Balanced Cost–Latency Initialization (BCLI) strategy in Algorithm 2 constructs a problem-aware placement vector $\boldsymbol{\Psi}$ by jointly considering static communication latency and deployment cost. Similar to LACI, cloudlets are processed according to the ordered sequence $\boldsymbol{c}_{ord}$ and assigned to distinct candidate points. However, BCLI does not perform end-device assignment or enforce the complete CPP constraints during initialization; these tasks are subsequently handled by PBDA and the repair procedure.

The preference parameter α controls the relative importance of deployment cost and static communication latency in BCLI. In the balanced regime ($0.3 < \alpha < 0.8$), it provides the balanced component of the initial population. In the latency-focused regime ($0 \le \alpha \le 0.3$), it provides a smaller supplementary component alongside LACI-generated placements, according to the quotas in Table 6. Including these BCLI placements does not change the regime's latency-focused weighting policy.

BCLI initializes the placement vector as $\boldsymbol{\Psi} := \mathbf{0}_{1\times M}$ and the set of unoccupied candidate points as $\mathcal{P}_{uo} := \mathcal{P}$. The maximum static communication latency $t_{\mathrm{S}}^{\max}$ is obtained from Equation (35) and is used by the normalized latency score defined in Equation (36).

For each cloudlet c_j and unoccupied candidate point $p_k \in \mathcal{P}_{uo}$, the set of end devices within the communication radius of c_j is defined as

$$\mathcal{E}_{jk}^{\text{cov}} = \left\{ e_i \in \mathcal{E} \mid d_{ik} \leq r_{c_j} \right\}, \qquad n_{jk}^{\text{cov}} = \left| \mathcal{E}_{jk}^{\text{cov}} \right|. \tag{43}$$

The set $\mathcal{E}_{jk}^{\text{cov}}$ is used only to evaluate the candidate placement and does not represent a final end-device assignment. Consequently, the same device may contribute to the evaluation of multiple cloudlet–candidate-point pairs.

For the current cloudlet c_j, the minimum deployment cost among the unoccupied candidate points is

$$C_{\min_j} = \min_{p_h \in \mathcal{P}_{uo}} C_{jh}. \tag{44}$$

Each candidate point providing nonzero coverage is then evaluated using the composite score

$$s_{jk} = (1-\alpha)\frac{1}{n_{jk}^{\text{cov}}} \sum_{e_i \in \mathcal{E}_{jk}^{\text{cov}}} s_{ik}^{\text{LT}} + \alpha \frac{C_{\min_j}}{C_{jk}}, \qquad n_{jk}^{\text{cov}} > 0. \tag{45}$$

The first term represents the average normalized static communication-latency score of the devices covered by (c_j, p_k). Static communication latency is used because the processing and queueing components of the complete latency depend on the final assignment and are therefore unavailable during initialization. The second term represents deployment-cost attractiveness: it equals 1 for the least expensive available candidate point and decreases as the deployment cost increases. Since both components are normalized to $[0, 1]$, the parameter α directly controls their relative influence, assigning weight $1 - \alpha$ to latency and α to deployment cost.

The eligible candidate-point index set for cloudlet c_j is defined as

$$\mathcal{K}_j = \left\{ k \mid p_k \in \mathcal{P}_{uo},\ \mathcal{E}_{jk}^{\text{cov}} \neq \emptyset \right\}. \tag{46}$$

If $\mathcal{K}_j \neq \emptyset$, BCLI selects the candidate point with the largest composite score, resolving ties by the smallest candidate-point index:

$$\nu_j = \min \left(\arg\max_{k \in \mathcal{K}_j} s_{jk} \right). \tag{47}$$

The selected placement is recorded as $\psi_{\nu_j} := j$, after which p_{ν_j} is removed from $\mathcal{P}_{uo}$. This prevents more than one cloudlet from being assigned to the same candidate point. If $\mathcal{K}_j = \emptyset$, cloudlet c_j remains undeployed during BCLI. Repeating this procedure for the cloudlets in $\boldsymbol{c}_{ord}$ produces an informed initial placement that reflects the selected cost–latency preference and is subsequently processed by the common assignment, repair, and optimization stages.

5.1.3. Cost-Focused Initialization (CFI) The Cost-Focused Initialization (CFI) algorithm 3 implements a greedy deployment strategy tailored to scenarios where cost constitutes the dominant factor in the objective function, particularly when $\alpha \geq 0.8$. For each cloudlet, the algorithm systematically assesses all currently unoccupied candidate points, identifying the number of covered end devices within the communication radius of the cloudlet. The eligible candidate-point index set $\mathcal{K}_j$ is constructed according to Equation (46). At every candidate point that provides nonzero coverage, CFI calculates the coverage-to-cost score as

$$h_{jk} = \frac{n_{jk}^{cov}}{C_{jk}}, \qquad k \in \mathcal{K}_j. \tag{48}$$

Here, n_{jk}^{cov} denotes the number of end devices covered by cloudlet c_j at candidate point p_k, and C_{jk} denotes the corresponding deployment cost. Thus, a larger value of h_{jk} indicates that the candidate placement provides greater coverage relative to its deployment cost. This coverage measure is used for initialization and does not represent a final end-device assignment. Therefore, an end device may contribute to the evaluation of multiple candidate placements.

When $\mathcal{K}_j \neq \emptyset$, CFI selects the candidate point having the maximum coverage-to-cost score. If several candidate points have the same maximum score, the smallest candidate-point index is selected:

$$\nu_j = \min \left(\arg\max_{k \in \mathcal{K}_j} h_{jk} \right), \qquad \mathcal{K}_j \neq \emptyset. \tag{49}$$

Algorithm 2: Balanced Cost–Latency Initialization (BCLI)

Input: $\mathcal{E}$, $\mathcal{C}$, ordered cloudlet sequence $\boldsymbol{c}_{ord}$, $\mathcal{P}$, $\boldsymbol{C}$, $\mathcal{D}$, $\{r_{c_j}\}$, $\boldsymbol{t}_{\mathrm{S}}$, and $\alpha \in [0, 0.8)$.
Output: Placement vector $\boldsymbol{\Psi}$.
// Initialize the placement state:
Initialize $\boldsymbol{\Psi} := \mathbf{0}_{1\times M}$ and $\mathcal{P}_{uo} := \mathcal{P}$
Obtain $t_{\mathrm{S}}^{\max}$ from Equation (35)
// Evaluate each cloudlet at the available candidate points:
foreach $c_j \in \boldsymbol{c}_{ord}$ **do**
  if $\mathcal{P}_{uo} \neq \emptyset$ **then**
    Compute $C_{\min_j}$ using Equation (44)
    foreach $p_k \in \mathcal{P}_{uo}$ **do**
      Construct $\mathcal{E}_{jk}^{\mathrm{cov}}$ and n_{jk}^{cov} using Equation (43)
      if $n_{jk}^{\mathrm{cov}} > 0$ **then**
        Compute s_{jk} using Equation (45)
    // Select and commit the best eligible placement:
    Construct $\mathcal{K}_j$ using Equation (46)
    if $\mathcal{K}_j \neq \emptyset$ **then**
      Select ν_j using Equation (47)
      Set $\psi_{\nu_j} := j$
      Set $\mathcal{P}_{uo} := \mathcal{P}_{uo} \setminus \{p_{\nu_j}\}$

If $\mathcal{K}_j = \emptyset$, cloudlet c_j remains undeployed during CFI. Cloudlets are examined according to the supplied order $\boldsymbol{c}_{ord}$, while available candidate points are examined in ascending index order. This process generates a cost-aware initial placement vector for the subsequent search process. The placement serves only as a starting point; final end-device assignment, constraint enforcement, and solution repair are performed by the subsequent assignment and repair stages.

Algorithm 3: Cost-Focused Initialization (CFI)

Input: $\mathcal{E}$, $\mathcal{C}$, ordered cloudlet sequence $\boldsymbol{c}_{ord}$, $\mathcal{P}$, $\boldsymbol{C}$, $\mathcal{D}$, and $\{r_{c_j}\}$.
Output: Placement vector $\boldsymbol{\Psi}$.
// Initialize the placement and available candidate points:
Initialize $\boldsymbol{\Psi} := \mathbf{0}_{1\times M}$ and $\mathcal{P}_{uo} := \mathcal{P}$
// Evaluate each cloudlet and construct its eligible candidate-point set:
foreach $c_j \in \boldsymbol{c}_{ord}$ **do**
  foreach $p_k \in \mathcal{P}_{uo}$ *in ascending index order* **do**
    // Obtain the end devices covered by the candidate placement:
    Construct $\mathcal{E}_{jk}^{cov}$ and obtain n_{jk}^{cov} using Equation (43)
    // Evaluate a candidate point providing nonzero coverage:
    if $n_{jk}^{cov} > 0$ **then**
      Compute h_{jk} using Equation (48)
  Construct $\mathcal{K}_j$ using Equation (46)
  // Assign the cloudlet when a candidate point has been selected:
  if $\mathcal{K}_j \neq \emptyset$ **then**
    Select ν_j using Equation (49)
    Place c_j at p_{ν_j} by setting $\psi_{\nu_j} := j$
    Remove p_{ν_j} from $\mathcal{P}_{uo}$

5.2. Priority-Based Device Assignment

After constructing a placement vector $\boldsymbol{\Psi}$, the Priority-Based Device Assignment (PBDA) Algorithm 4 assigns end devices to the deployed cloudlets while enforcing communication-range and resource-capacity requirements.

PBDA follows a hardest-to-cover-first strategy, giving priority to devices with fewer reachable cloudlet–candidate-point pairs. This reduces the likelihood that highly constrained devices remain uncovered after available cloudlet resources have been consumed by devices with more assignment alternatives.

PBDA first identifies the cloudlets currently deployed in the placement vector $\mathbf{\Psi}$. For each candidate point p_k, if $\psi_k \neq 0$ and $j \neq 0$, cloudlet c_j is deployed at p_k, and the corresponding cloudlet–candidate-point pair (j,k) is included in the deployed-pair set $\mathcal{P}_{dep}$:

$$\mathcal{P}_{dep} = \{(j,k) \mid \psi_k = j,\ j \neq 0,\ k \in \{1,\ldots,M\}\}. \tag{50}$$

For each end device e_i, PBDA constructs the set of deployed pairs within its communication range:

$$\mathcal{S}_i = \{(j,k) \in \mathcal{P}_{dep} : d_{ik} \leq r_{c_j}\}. \tag{51}$$

Devices for which $\mathcal{S}_i = \emptyset$ are added directly to the uncovered-device set $\mathcal{E}_{uc}$. The remaining devices are processed in ascending order of $|\mathcal{S}_i|$, with ties resolved by device index, so that devices having the fewest assignment alternatives are considered first. When device e_i is processed, its reachable pairs are filtered according to the remaining processor, memory, and storage capacities of the corresponding cloudlets:

$$V_i = \left\{(j,k) \in \mathcal{S}_i \mid \mathcal{U}_{c_j}[r] + \mathcal{R}_{e_i}[r] \leq \mathcal{R}_{c_j}[r],\ \forall r \in \{1,2,3\}\right\}. \tag{52}$$

If $V_i = \emptyset$, the device remains uncovered. Otherwise, the feasible pairs are ranked primarily by static communication latency,

$$v_1(j,k) = t_{\mathrm{S}_{ik}} = t_{\mathrm{T}_i} + t_{\mathrm{P}_{ik}}, \tag{53}$$

and secondarily by the current normalized resource utilization of cloudlet c_j:

$$v_2(j) = \frac{1}{3}\sum_{r=1}^{3} \frac{\mathcal{U}_{c_j}[r]}{\mathcal{R}_{c_j}[r]}. \tag{54}$$

PBDA selects the feasible pair $(j^\star, k^\star)$ with the lowest static communication latency. Ties are resolved successively by the lowest cloudlet utilization, the smallest candidate-point index k, and finally the smallest cloudlet index j. The assignment $(i, j^\star, k^\star)$ is then added to Ω, and the resource usage of the selected cloudlet is updated as

$$\mathcal{U}_{c_{j^\star}}[r] := \mathcal{U}_{c_{j^\star}}[r] + \mathcal{R}_{e_i}[r], \qquad \forall r \in \{1,2,3\}. \tag{55}$$

After each assignment, PBDA identifies highly utilized cloudlets using

$$\max_{r=1}^{3}\left(\frac{\mathcal{U}_{c_{j^\star}}[r]}{\mathcal{R}_{c_{j^\star}}[r]}\right) \geq \tau_{\mathrm{up}}. \tag{56}$$

When this condition is satisfied, $(j^\star, k^\star)$ is added to the upgrade-candidate set Q_{up} for subsequent processing in section 5.3.

Termination and conditional feasibility. PBDA terminates because it processes finite sets of devices and deployed cloudlet–candidate-point pairs. Starting from a structurally valid placement and zero resource usage, each device is assigned at most once, and each accepted assignment satisfies the communication-range and remaining-capacity checks. Updating resource usage after each assignment preserves these capacity conditions. Complete coverage is obtained if every device has at least one reachable, capacity-feasible pair when its turn is processed. Otherwise, the unassigned devices are recorded for repair; PBDA does not guarantee finding a complete assignment whenever one exists.

5.3. Coverage and Resource Repair (CRR)

After executing the PBDA Algorithm 4, a candidate placement may still contain uncovered end devices or highly utilized cloudlets that could benefit from additional capacity. The Coverage and Resource Repair (CRR) stage

Algorithm 4: Priority-Based Device Assignment (PBDA)

Input: Placement vector $\mathbf{\Psi}$, $\mathcal{E}$, $\{\mathcal{R}_{e_i}\}$, $\{\mathcal{R}_{c_j}\}$, $\mathcal{D}$, $\{r_{c_j}\}$, $\boldsymbol{t}_{\mathrm{S}}$, and τ_{up}.
Output: Ω, $\mathcal{E}_{uc}$, $\{\mathcal{U}_{c_j}\}$, and Q_{up}.

// Construct the deployed cloudlet–candidate-point set:
Construct $\mathcal{P}_{dep}$ using Equation (50)
// Initialize the assignment and resource-usage state:
Initialize $\mathcal{U}_{c_j} := \mathbf{0}_{1\times 3}$ for every $(j,k) \in \mathcal{P}_{dep}$
Initialize $\Omega := \emptyset$ and the upgrade-candidate set $Q_{\mathrm{up}} := \emptyset$
Initialize $\mathcal{E}_{uc} := \emptyset$ and $\mathcal{E}_{reach} := \emptyset$
// Construct the reachable deployment-pair set of each end device:
foreach $e_i \in \mathcal{E}$ **do**
    Construct $\mathcal{S}_i$ using Equation (51)
    if $\mathcal{S}_i = \emptyset$ **then**
        Add e_i to $\mathcal{E}_{uc}$
        continue
    Add $(i, \mathcal{S}_i)$ to $\mathcal{E}_{reach}$
// Order the reachable end devices from hardest to easiest:
Order the pairs in $\mathcal{E}_{reach}$ by ascending $|\mathcal{S}_i|$, breaking ties by device index i, and store them in $\mathcal{E}_{ord}$
// Assign each reachable end device to a capacity-feasible pair:
foreach $(i, \mathcal{S}_i) \in \mathcal{E}_{ord}$ **do**
    Construct V_i using Equation (52)
    if $V_i = \emptyset$ **then**
        Add e_i to $\mathcal{E}_{uc}$
        continue
    // Select the feasible deployment pair using the ordered ranking criteria:
    Select $(j^\star, k^\star) \in V_i$ by prioritizing the lowest $v_1(j,k)$ from Equation (53), then the lowest $v_2(j)$ from Equation (54), the smallest k, and finally the smallest j
    Add $(i, j^\star, k^\star)$ to Ω
    Update the usage of cloudlet $c_{j^\star}$ using Equation (55)
    // Flag the selected pair when a resource reaches the upgrade threshold:
    if *the selected pair satisfies the upgrade condition in Equation* (56) **then**
        Add $(j^\star, k^\star)$ to Q_{up}

improves the existing candidate through localized modifications while preserving valid placement and assignment decisions whenever possible. CRR first processes highly utilized cloudlets by attempting feasible upgrades to higher-capacity cloudlets at their current candidate points. Any additional capacity created by successful upgrades is then used to reconsider and assign previously uncovered devices. If uncovered devices remain, CRR searches the unoccupied candidate points for suitable repair locations and attempts to deploy an available cloudlet that can provide the required coverage and resource capacity. When no feasible undeployed cloudlet can serve the targeted devices at a new location, CRR applies a fallback strategy by considering an already occupied candidate point and attempting a feasible cloudlet upgrade at that location. In this manner, CRR progressively improves coverage and resource availability without reconstructing the complete placement. The repair process proceeds as follows:

1. **Process highly utilized cloudlets.** CRR first examines the cloudlet–candidate-point pairs identified by PBDA as highly utilized. Whenever possible, the deployed cloudlet is replaced at the same candidate point by an available higher-tier cloudlet, provided that the replacement can accommodate the current resource usage and preserve coverage of all devices already assigned to that location. The existing assignments and resource usage are then transferred to the upgraded cloudlet.

2. **Reassign uncovered devices after successful upgrades.** The additional capacity provided by successful upgrades is used to accommodate currently uncovered end devices. These devices are reconsidered according to the PBDA assignment rules, while communication-range and resource-capacity requirements remain enforced.

3. **Identify candidate points for coverage repair.** If uncovered devices remain, CRR examines the currently unoccupied candidate points and determines which of them could potentially provide coverage when paired

with an available undeployed cloudlet. Candidate points that cannot cover any remaining uncovered device are excluded from further consideration.

4. **Select the most suitable repair location.** For each eligible candidate point, CRR calculates a repair score based on its potential to cover the remaining uncovered devices, the proximity of those devices, and the required deployment cost. A larger score represents a location that can potentially cover more devices with shorter distances and lower deployment cost. The candidate point having the highest repair score is selected as the repair location.

5. **Deploy a feasible undeployed cloudlet.** For the selected candidate point, CRR identifies the undeployed cloudlets capable of covering the targeted device group while satisfying the group's combined processor, memory, and storage requirements. If multiple cloudlets are feasible, the cloudlet with the lowest deployment cost is selected. The placement, device assignments, resource usage, uncovered-device set, undeployed-cloudlet set, and upgrade candidates are then updated accordingly.

6. **Apply fallback repair when a new deployment is infeasible.** If no undeployed cloudlet can serve the targeted device group at the selected point, CRR attempts to exploit the existing placement. An occupied candidate point close to the uncovered-device group is selected, and its deployed cloudlet is considered for replacement by an available higher-tier cloudlet. Following a successful upgrade, uncovered devices are assigned whenever both communication-range and residual resource-capacity requirements are satisfied.

7. **Apply radius-extension fallback when necessary.** Radius extension is attempted when uncovered devices remain and preceding repair actions cannot improve coverage. CRR considers extending a deployed cloudlet's radius or deploying an available cloudlet at an unoccupied point with the required extension. Each trial is reassessed by PBDA using the updated communication radius and the cloudlet's resource capacities. Only trials that reduce the number of uncovered devices are eligible. Among these eligible trials, CRR prioritizes fewer uncovered devices, then lower extension cost and lower latency; remaining ties are resolved by candidate-point and cloudlet indices.

8. **Repeat the repair process when necessary.** The coverage-repair procedure is repeated while uncovered devices remain and further repair actions are possible. Repair terminates when all devices are covered, no suitable repair location or upgrade is available, no additional device can be assigned, or the predefined repair limit is reached. CRR cannot repeat indefinitely: it stops when repair succeeds, no further progress is possible, or the maximum number of repair attempts is reached.

9. **Return the repaired candidate.** The resulting placement, assignment state, and resource-usage information are returned for subsequent evaluation. Any end devices that remain uncovered after CRR are retained as residual infeasibility and are handled by the constraint-violation penalty during fitness evaluation.

Effective-Parameter Evaluation after Radius Extension. The following expressions specify how the nominal model is evaluated after radius extension. Let r_{c_j} denote the nominal radius and $r_{c_j}^{\mathrm{eff}}$ the current effective radius. For an uncovered device targeted by an extension trial,

$$r_{c_j}^{\mathrm{eff,new}} = \max\left(r_{c_j}^{\mathrm{eff,current}}, d_{ik} + \varepsilon_r\right), \qquad \varepsilon_r = 10^{-6}.$$

The extension relative to the nominal radius and its additional deployment cost are

$$\delta_j = r_{c_j}^{\mathrm{eff}} - r_{c_j} \geq 0, \qquad C_{jk}^{\mathrm{ext}} = C_{jk}\frac{\delta_j}{r_{c_j}},$$

so that

$$C_{jk}^{\mathrm{eff}} = C_{jk} + C_{jk}^{\mathrm{ext}}.$$

When no extension occurs, $\delta_j = 0$, so the effective radius and cost equal their nominal values. Radius extension is an abstract coverage–cost modeling assumption; the proportional surcharge is expressed in virtual cost units rather than measured financial or radio-engineering costs.

For a repaired candidate, Equation (18) is evaluated using effective deployment costs:

$$f_C^{\text{eff}} = \sum_{j=1}^{L} \sum_{k=1}^{M} C_{jk}^{\text{eff}} y_{jk}.$$

The seeding strategies retain their nominal-cost scoring rules. Effective costs are used when evaluating the resulting repaired candidate.

For a repaired candidate, the communication-range condition in Constraint (20c) uses the effective radius:

$$w_{ijk} \le \min\Big(y_{jk}, \mathbb{1}[d_{ik} \le r_{c_j}^{\text{eff}}]\Big).$$

Resource-capacity and deployment-structure constraints remain enforced. A radius-extended candidate is assessed under this repair-aware communication-range condition, not under the unchanged nominal-radius condition.

The scalar objective in Equation (29) is evaluated using the repaired candidate's effective deployment cost:

$$F_\alpha^{\text{eff}} = \omega_C(\alpha) \frac{f_C^{\text{eff}}}{f_C^{\max}} + \omega_L(\alpha) \frac{f_L}{f_L^{\max}}.$$

The same fixed normalization references are used before and after repair. The latency formula remains unchanged, but its value is recomputed from the updated assignment when necessary.

Similarly, the service-link penalty in Equation (70) is evaluated by substituting $r_{c_j}^{\text{eff}}$ for the nominal radius. Thus, extension changes the candidate's communication-range test; it does not remove capacity, deployment, or assignment-consistency checks.

5.4. Search Space Generation

The Population Initialization (PopInit) procedure in Algorithm 5 constructs the initial population $\mathbf{\Psi}_p$ supplied to the subsequent GA or PSO update stage. Rather than relying exclusively on random initialization, PopInit combines coverage-oriented, balanced or cost-oriented, and random seeds to provide both problem-aware starting solutions and population diversity. The required population size is defined as:

$$N_{pop} = N_{cov} + N_{bla} + N_{ran}, \tag{57}$$

where N_{cov} represents solutions focused on coverage, N_{bla} refers to balanced or greedy cost-aware solutions, and N_{ran} stands for randomly created solutions. These three quotas are not fixed constants; they are selected according to the preference value α. The corresponding quota allocations and acceptance thresholds are reported in Table 6. In latency-focused settings, more seeds are allocated to coverage-oriented placement. In balanced settings, coverage and balanced seeds receive moderate shares. In cost-focused settings, more seeds are allocated to the greedy cost-aware component. The population is initially set as an empty collection, expressed as $\mathbf{\Psi}_p = \emptyset$.

The algorithm initiates a main generation loop, which persists until one of two termination criteria is satisfied: the attainment of the specified population size or the exhaustion of the maximum allowable number of generation attempts. Specifically, the loop concludes when either $|\mathbf{\Psi}_p| = N_{pop}$ or $a = A_{max}$, with a denoting the current attempt count and A_{max} the upper bound on attempted solutions. The attempt-scaling factor $\lambda_A > 1$ defines this limit as $A_{max} = \lambda_A N_{pop}$. The coverage stage ends at $A_{cov} = \lambda_A N_{cov}$ attempts, and the balanced/cost-oriented stage ends at the cumulative limit $A_{cov} + A_{bla}$, where $A_{bla} = \lambda_A N_{bla}$. If either target is not reached within its limit, PopInit proceeds to the next stage, and the random stage attempts to fill any remaining positions. The values of λ_A and s_0 are reported in Table 4.

LACI, BCLI, and CFI are greedy initialization algorithms; therefore, their resulting placement vectors may depend on the order in which cloudlets are processed. Reusing the same order in every attempt could repeatedly

produce the same placement vector. To generate varied but reproducible solutions, PopInit creates a new cloudlet-processing order for each attempt. Let

$$\text{seededShuffle}(\mathcal{C}, s)$$

denote an operation that returns all cloudlets in $\mathcal{C}$ exactly once in an order determined by seed s. At attempt a, PopInit calculates

$$\boldsymbol{c}_{ord} := \text{seededShuffle}(\mathcal{C}, s_0 + a),$$

Here, s_0 denotes the instance-specific random seed. Consequently, different attempts may produce different processing orders, whereas repeating the experiment with the same s_0 reproduces the same sequence of orders. The selected initializer then processes the cloudlets according to $\boldsymbol{c}_{ord}$.

5.4.1. Coverage-Oriented Population Generation The first part of the population is generated using the LACI algorithm, which targets N_{cov} coverage-oriented solutions. These solutions are designed to serve a larger number of end devices and usually tend to reduce latency, because they focus on selecting cloudlet placements that can cover more end devices efficiently. These solutions are then inserted into the population $\boldsymbol{\Psi}_p$.

5.4.2. Balanced or Cost-Oriented Population Generation After the coverage-oriented stage reaches its target or its attempt limit, the algorithm proceeds to generate the balanced or cost-oriented component, whose target size is denoted by N_{bla}. The generation strategy for this component is controlled by the preference parameter α.

For balanced or cost-oriented seed generation, if $\alpha \geq 0.8$, the algorithm invokes the $\text{CFI}()$ algorithm to generate cost-oriented initial solutions. Otherwise, the $\text{BCLI}()$ algorithm is used to generate initial solutions that balance deployment cost and latency. This stage continues until the number of accepted structured solutions satisfies:

$$|\boldsymbol{\Psi}_p| \geq N_{cov} + N_{bla}. \tag{58}$$

5.4.3. Random Population Generation The remaining population spaces, including any shortfall from the preceding stages, are filled by candidate solutions generated randomly while $|\boldsymbol{\Psi}_p| < N_{pop}$. At random-generation attempt a, PopInit independently creates a reproducible cloudlet order and candidate-point order:

$$\boldsymbol{c}_{rnd} := \text{seededShuffle}(\mathcal{C}, s_0 + 2a),$$

$$\boldsymbol{p}_{rnd} := \text{seededShuffle}(\mathcal{P}, s_0 + 2a + 1).$$

PopInit pairs the first cloudlet in $\boldsymbol{c}_{rnd}$ with the first candidate point in $\boldsymbol{p}_{rnd}$, the second cloudlet with the second candidate point, and so forth, until all cloudlets are placed. Because $L < M$, every cloudlet is assigned to a distinct candidate point, while the remaining candidate points stay unoccupied. The shuffle seeds are derived from s_0 and the attempt number a, allowing the same random placements to be reproduced. When the search initializes its population solely utilizing LACI, CFI, or BCLI, its capacity to investigate various regions of the search space may be constrained. Consequently, the inclusion of random candidate solutions is implemented to augment population diversity and mitigate the likelihood of premature convergence.

5.4.4. Feasibility Repair and Acceptance Test Each generated candidate placement vector is first processed using PBDA algorithm 4 to determine its assignment set, uncovered end devices, resource usage, and upgrade candidates. If any end devices remain uncovered or $Q_{\text{up}} \neq \emptyset$, CRR stage is applied to process the upgrade candidates and improve the candidate. PBDA is then applied again to reconstruct the assignment set, resource-usage vectors, uncovered-device set, and upgrade-candidate set for the repaired placement. This revalidation call uses the same four outputs as the first PBDA call. The resulting Q_{up} records highly utilized pairs identified after repair but does not affect the seed-acceptance decision because the revalidated assignments satisfy the resource-capacity constraints and acceptance is determined by the covered-device ratio. Based on the final revalidated assignment outcome, the candidate is admitted into the initial population $\boldsymbol{\Psi}_p$ only if it satisfies:

$$\frac{|\mathcal{E}| - |\mathcal{E}_{uc}|}{|\mathcal{E}|} \geq \tau_{init}, \tag{59}$$

where $|\mathcal{E}|$ is the total number of end devices, $|\mathcal{E}_{uc}|$ is the number of uncovered end devices, and τ_{init} is the normal minimum coverage ratio required for accepting a candidate solution. PBDA returns Q_{up} to CRR, which processes the flagged pairs before coverage repair. The acceptance decision uses the uncovered set $\mathcal{E}_{uc}$ after assignment and repair. If equation (59) is not satisfied, the candidate solution is rejected, and a new candidate is generated in the next attempt.

5.4.5. Emergency Completion Stage If the main initialization loop terminates due to reaching the maximum number of attempts A_{max}, while the initial population remains incomplete, the algorithm invokes an emergency completion stage. This fallback mechanism is activated when the accepted population remains smaller than the required population size:

$$|\mathbf{\Psi}_p| < N_{pop}. \tag{60}$$

Under this condition, the number of missing candidate solutions is computed as:

$$N_{emg} = N_{pop} - |\mathbf{\Psi}_p|. \tag{61}$$

The emergency-attempt counter e is initialized to zero, and its maximum number of attempts is defined as

$$E_{max} = \lambda_A \max(1, N_{emg}). \tag{62}$$

The emergency stage terminates when the population reaches N_{pop} solutions or when $e = E_{max}$.

The emergency stage attempts to generate up to N_{emg} additional distinct solutions using the cost-oriented CFI algorithm. Unlike the main initialization loop, the emergency completion stage applies a relaxed acceptance criterion. Specifically, an emergency candidate solution is accepted if its covered-device ratio satisfies:

$$\frac{|\mathcal{E}| - |\mathcal{E}_{uc}|}{|\mathcal{E}|} \geq \tau_{emg}, \tag{63}$$

where τ_{emg} denotes the emergency coverage threshold. This threshold is used only in the emergency completion stage and is defined as a relaxed version of the normal initialization threshold τ_{init}, such that:

$$\tau_{emg} < \tau_{init}. \tag{64}$$

This relaxation enables the algorithm to accept candidate solutions with somewhat lower coverage quality during the fallback phase. While these solutions might not meet the more stringent initialization requirements, including them helps complete the initial population before the selected iterative update search begins.

If the population remains incomplete after emergency initialization, PopInit attempts final completion using random candidates. The final-completion attempt limit is

$$F_{\max} = \max(F_{\min}, \lambda_A N_{pop}^2).$$

Here, $F_{\min}$ denotes the minimum final-completion attempt budget; its experimental value is reported in Table 4. At attempt f, cloudlet and candidate-point orders use seeds $s_0 + 2f$ and $s_0 + 2f + 1$, respectively. Previously generated placement vectors are skipped. Each new candidate undergoes PBDA, CRR when required, and PBDA revalidation, and is accepted only if its repaired placement vector is not already present. The coverage acceptance threshold is not applied during final completion; remaining infeasibility enters the fitness penalties. Completion stops when the population reaches N_{pop} or the attempt limit is exhausted. If the population remains incomplete, initialization reports failure.

For readability, PopInit writes calls to LACI, BCLI, CFI, PBDA, and CRR in compact form. The input and output definitions of LACI, BCLI, CFI, and PBDA are provided in their corresponding algorithms, while the CRR procedure and its returned repair state are described in Subsection 5.3. PopInit shows only the changing placement vector and the returned values needed for population construction.

Algorithm 5: Population Initialization (PopInit)

Input: α, N_{pop}, N_{cov}, N_{bla}, τ_{init}, τ_{emg}, λ_A, s_0, $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{P}$.
Output: Population $\mathbf{\Psi}_p = \{\mathbf{\Psi}^{(1)}, \mathbf{\Psi}^{(2)}, \ldots, \mathbf{\Psi}^{(N_{pop})}\}$.
// Initialize the population, quotas, and attempt limits:
Initialize $\mathbf{\Psi}_p := \emptyset$
Set $A_{max} := \lambda_A N_{pop}$
Set $A_{cov} := \lambda_A N_{cov}$ and $A_{bla} := \lambda_A N_{bla}$
Initialize $a := 0$
// Generate candidate seeds until the population or attempt limit is reached:
while $|\mathbf{\Psi}_p| < N_{pop}$ *and* $a < A_{max}$ **do**
    Increase a by 1
    // Create a shuffled cloudlet-processing order:
    Set $\boldsymbol{c}_{ord} := \text{seededShuffle}(\mathcal{C}, s_0 + a)$
    // Select the seed-generation strategy:
    if $|\mathbf{\Psi}_p| < N_{cov}$ *and* $a \leq A_{cov}$ **then**
        Set $\mathbf{\Psi}_{seed} := \text{LACI}(\boldsymbol{c}_{ord})$
    else if $|\mathbf{\Psi}_p| < N_{cov} + N_{bla}$ *and* $a \leq A_{cov} + A_{bla}$ **then**
        if $\alpha \geq 0.8$ **then**
            Set $\mathbf{\Psi}_{seed} := \text{CFI}(\boldsymbol{c}_{ord})$
        else
            Set $\mathbf{\Psi}_{seed} := \text{BCLI}(\boldsymbol{c}_{ord})$
    else
        // Construct the random placement seed:
        Initialize $\mathbf{\Psi}_{seed} := \mathbf{0}_{1\times M}$
        Set $\boldsymbol{c}_{rnd} := \text{seededShuffle}(\mathcal{C}, s_0 + 2a)$
        Set $\boldsymbol{p}_{rnd} := \text{seededShuffle}(\mathcal{P}, s_0 + 2a + 1)$
        Initialize $q := 1$
        foreach $c_j \in \boldsymbol{c}_{rnd}$ **do**
            Set $p_k := \boldsymbol{p}_{rnd}[q]$
            Set $\mathbf{\Psi}_{seed}[k] := j$
            Increase q by 1
    // Assign devices, repair the seed, and revalidate it:
    Apply PBDA, CRR when required, and PBDA revalidation to $\mathbf{\Psi}_{seed}$
    Compute ρ_{seed} using the covered-device ratio in Equation (59)
    // Accept a sufficiently covered and nonduplicate seed:
    if $\rho_{seed} \geq \tau_{init}$ *and* $\mathbf{\Psi}_{seed} \notin \mathbf{\Psi}_p$ **then**
        Set $\mathbf{\Psi}_p := \mathbf{\Psi}_p \cup \{\mathbf{\Psi}_{seed}\}$
// Initialize the emergency completion stage:
Compute N_{emg} and E_{max} using Equations (61) and (62)
Initialize $e := 0$
// Generate bounded emergency seeds until the population is complete:
while $|\mathbf{\Psi}_p| < N_{pop}$ *and* $e < E_{max}$ **do**
    Increase e by 1
    Set $\boldsymbol{c}_{ord} := \text{seededShuffle}(\mathcal{C}, s_0 + A_{max} + e)$
    Set $\mathbf{\Psi}_{seed} := \text{CFI}(\boldsymbol{c}_{ord})$
    // Assign devices, repair the emergency seed, and revalidate it:
    Apply PBDA, CRR when required, and PBDA revalidation to $\mathbf{\Psi}_{seed}$
    Compute ρ_{seed} using the covered-device ratio in Equation (59)
    // Accept a relaxed and nonduplicate emergency seed:
    if $\rho_{seed} \geq \tau_{emg}$ *and* $\mathbf{\Psi}_{seed} \notin \mathbf{\Psi}_p$ **then**
        Set $\mathbf{\Psi}_p := \mathbf{\Psi}_p \cup \{\mathbf{\Psi}_{seed}\}$
// Complete the population using random initialization:
Initialize $f := 0$
Set $F_{\max} := \max(F_{\min}, \lambda_A N_{pop}^2)$
while $|\mathbf{\Psi}_p| < N_{pop}$ *and* $f < F_{\max}$ **do**
    Increase f by 1
    Generate a structurally valid random placement vector $\mathbf{\Psi}_{seed}$ using shuffled cloudlet and candidate-point orders determined by s_0 and f, ensuring that $\mathbf{\Psi}_{seed}$ has not been generated previously in this stage
    Apply PBDA, CRR when required, and PBDA revalidation to $\mathbf{\Psi}_{seed}$
    if $\mathbf{\Psi}_{seed} \notin \mathbf{\Psi}_p$ **then**
        Set $\mathbf{\Psi}_p := \mathbf{\Psi}_p \cup \{\mathbf{\Psi}_{seed}\}$
if $|\mathbf{\Psi}_p| < N_{pop}$ **then**
    Report initialization failure

5.5. Fitness Evaluation with Residual Constraint-Violation Penalties

After PBDA assignment and CRR repair, each candidate is evaluated using the scalar cost–latency objective. For candidates with radius extension, cost and service-link evaluation follow the effective-parameter substitutions defined in Subsection 5.3. Here, $F_\alpha^{(n)}$ denotes this candidate-specific evaluation. Residual constraint violations are then incorporated through the penalty term:

$$\mathcal{F}\left(\mathbf{\Psi}^{(n)}\right) = F_\alpha^{(n)} + P_{\text{total}}\left(\mathbf{\Psi}^{(n)}\right), \tag{65}$$

where $\mathcal{F}(\mathbf{\Psi}^{(n)})$ is the fitness used to rank candidate solutions and $P_{\text{total}}(\mathbf{\Psi}^{(n)})$ measures any constraint violations that remain after placement construction, assignment, and repair. The total residual penalty combines coverage, resource-capacity, structural, service-link, and assignment-consistency violations:

$$\begin{aligned} P_{\text{total}}\left(\mathbf{\Psi}^{(n)}\right) = \lambda_{\text{cov}} V_{\text{cov}}\left(\mathbf{\Psi}^{(n)}\right) + \sum_{r=1}^{3} \lambda_{\text{cap},r} V_{\text{cap},r}\left(\mathbf{\Psi}^{(n)}\right) \\ + \lambda_{\text{struct}} V_{\text{struct}}\left(\mathbf{\Psi}^{(n)}\right) + \lambda_{\text{link}} V_{\text{link}}\left(\mathbf{\Psi}^{(n)}\right) + \lambda_{\text{cons}} V_{\text{cons}}\left(\mathbf{\Psi}^{(n)}\right). \end{aligned} \tag{66}$$

All penalty coefficients are positive. Separate coefficients $\lambda_{\text{cap},r}$ are used for the three resource dimensions because processor, memory, and storage violations may have different numerical scales.

The penalty coefficients remain fixed during optimization. Resource-capacity violations enter the penalty in their corresponding resource units, without additional normalization. Consequently, equal coefficient values do not imply equal contributions to fitness: each contribution also depends on the numerical magnitude of its residual violation.

Coverage Violation. The coverage violation is the number of end devices that remain uncovered after PBDA and CRR:

$$V_{\text{cov}}\left(\mathbf{\Psi}^{(n)}\right) = \left|\mathcal{E}_{uc}^{(n)}\right|. \tag{67}$$

This term corresponds to the complete-coverage requirement in Constraint (20b) and becomes zero when all end devices are successfully served.

Resource-Capacity Violation. In resource-indexed expressions, $r = 1, 2, 3$ refers to processor, memory, and storage, respectively. Radius, cloudlet category, and task size are separate attributes and are not included in these capacity calculations. For each resource dimension $r \in \{1, 2, 3\}$, the residual capacity violation measures the total assigned demand that exceeds the available capacity of the deployed cloudlets:

$$V_{\text{cap},r}\left(\mathbf{\Psi}^{(n)}\right) = \sum_{j=1}^{L} \max\left(0, \sum_{i=1}^{N}\sum_{k=1}^{M} w_{ijk}^{(n)} \mathcal{R}_{e_i}[r] - \mathcal{R}_{c_j}[r] \sum_{k=1}^{M} y_{jk}^{(n)}\right). \tag{68}$$

This term corresponds to the memory, storage, and processor capacity requirements in Constraints (20i)–(20k). If CRR replaces a cloudlet by a higher-tier cloudlet, the capacity $\mathcal{R}_{c_j}[r]$ represents the effective capacity of that repaired candidate.

Structural Deployment Violation. A structural violation occurs when more than one cloudlet occupies the same candidate point or when the same cloudlet is deployed at more than one candidate point. It is measured as

$$V_{\text{struct}}\left(\mathbf{\Psi}^{(n)}\right) = \sum_{k=1}^{M} \max\left(0, \sum_{j=1}^{L} y_{jk}^{(n)} - 1\right) + \sum_{j=1}^{L} \max\left(0, \sum_{k=1}^{M} y_{jk}^{(n)} - 1\right). \tag{69}$$

Accordingly, this penalty represents residual violations of Constraints (20l) and (20m).

Invalid Service-Link Violation. A service association is invalid if an end device is assigned to a cloudlet–candidate-point pair that is not deployed or lies outside the cloudlet's communication radius. The corresponding violation is

$$V_{\text{link}}\left(\mathbf{\Psi}^{(n)}\right) = \sum_{i=1}^{N}\sum_{j=1}^{L}\sum_{k=1}^{M} w_{ijk}^{(n)}\left(\mathbb{1}\left[y_{jk}^{(n)}=0\right] + \mathbb{1}\left[d_{ik} > r_{c_j}\right]\right), \tag{70}$$

where $\mathbb{1}[\cdot]$ denotes the indicator function. This term captures residual violations of the deployment and communication-range condition in Constraint (20c).

When radius extension is applied, this expression uses the effective radius defined in Subsection 5.3.

Assignment-Consistency Violation. The assignment-consistency violation verifies that the cloudlet-assignment variable x_{ij} and service-location variable z_{ik} remain consistent with the detailed service-association variable w_{ijk}:

$$\begin{aligned} V_{\text{cons}}\left(\mathbf{\Psi}^{(n)}\right) = &\sum_{i=1}^{N}\sum_{j=1}^{L}\left|x_{ij}^{(n)} - \sum_{k=1}^{M} w_{ijk}^{(n)}\right| + \sum_{i=1}^{N}\sum_{k=1}^{M}\left|z_{ik}^{(n)} - \sum_{j=1}^{L} w_{ijk}^{(n)}\right| \\ &+ \sum_{i=1}^{N}\max\left(0, \sum_{j=1}^{L} x_{ij}^{(n)} - 1\right) + \sum_{i=1}^{N}\left|\sum_{k=1}^{M} z_{ik}^{(n)} - 1\right|. \end{aligned} \tag{71}$$

This term corresponds primarily to the assignment relationships and uniqueness requirements in Constraints (20d), (20e), (20g), and (20h).

A candidate solution is fully feasible when all residual violation terms are zero, equivalently,

$$P_{\text{total}}\left(\mathbf{\Psi}^{(n)}\right) = 0. \tag{72}$$

Otherwise, the positive penalty increases $\mathcal{F}(\mathbf{\Psi}^{(n)})$, reducing the likelihood that an infeasible candidate is preferred solely because of an attractive cost–latency value. The resulting fitness therefore guides the search toward solutions that jointly achieve favorable deployment cost and service latency while satisfying the CPP feasibility requirements.

5.6. Unified GA–PSO Population Update Mechanisms within the LCRA Framework

After the initial population has been generated, assigned, repaired, and evaluated, LCRA employs a population-based metaheuristic to refine the candidate placement solutions iteratively. The population-update mechanism is deliberately separated from the problem-aware initialization and feasibility-handling components of the framework. Consequently, different metaheuristics can operate on the same placement representation while using the same PBDA assignment, CRR repair, latency model, and penalty-aware fitness evaluation.

In this study, Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) are considered as two alternative population-update mechanisms. Both receive the population produced by LCRA, generate an updated set of placement vectors, and return these placements to the common LCRA pipeline for reassignment and fitness evaluation. Thus, GA and PSO are treated as interchangeable search mechanisms rather than independent end-to-end cloudlet-placement methods.

5.6.1. Genetic Algorithm Update The Genetic Algorithm (GA) is used as an evolutionary search mechanism for refining the current population of cloudlet placements. In general, GA improves a population through selection and variation operations that combine and modify promising candidate solutions across successive generations [48]. Within LCRA, GA operates directly on the placement vectors supplied by the framework and produces a new population of candidate placements.

The GA uses the common PBDA assignment, CRR repair, and penalty-aware fitness procedures while updated candidates are constructed and selected, as well as during outer-framework evaluation. The implemented GA uses tournament selection, uniform crossover, mutation, and elitism. Mutation modifies placements through addition, movement, or swapping.

5.6.2. Particle Swarm Optimization Update Particle Swarm Optimization (PSO) provides an alternative population-update mechanism based on swarm intelligence. In general, PSO iteratively improves a population by guiding candidate solutions according to their previous search experience and information obtained from high-quality solutions identified by the swarm [49]. Since the CPP representation used in this work is discrete, PSO operates on the same cloudlet-placement vectors used throughout the LCRA framework.

The implemented discrete PSO uses current-particle, personal-best, and global-best influences, followed by mutation. These influences govern discrete placement-gene selection rather than a continuous velocity equation. As with GA, common assignment, repair, and fitness procedures are invoked during updated-candidate construction and selection, as well as during outer-framework evaluation. Both updaters use a duplicate-child attempt limit; duplicates may be accepted after this limit to complete the population.

Using GA and PSO under this unified pipeline enables the effect of problem-aware initialization to be examined under two different metaheuristic search mechanisms while avoiding differences in problem representation, assignment, repair, or fitness evaluation.

5.7. End-to-End Execution Process of the LCRA Framework

The previous subsections describe the individual components of the proposed LCRA framework. This subsection integrates these components into the complete execution process shown in Figure 3. No additional optimization mechanism is introduced at this stage; rather, the procedures defined in Subsections 5.1–5.6 are organized into a unified workflow that transforms a CPP instance into a final cloudlet-placement solution. The framework proceeds through seven main phases.

Phase 1: CPP Instance Preparation. The process begins by loading the CPP instance, including the end devices, available cloudlets, candidate points, deployment costs, resource capacities and demands, distance information, and latency parameters. The preference parameter α and the remaining initialization, repair, and search parameters are also specified. These data provide the common problem representation used throughout all subsequent stages.

Phase 2: Problem-Aware Search-Space Construction. PopInit constructs the initial placement population using LACI, BCLI or CFI according to the selected preference regime, and random placements for population diversity. Emergency completion is applied when necessary to obtain the required population size. The complete seed-generation, feasibility-handling, and acceptance procedure is defined in Subsection 5.4.

Phase 3: Device Assignment. After PopInit returns the completed population, PBDA, described in Subsection 5.2, is applied to each initialized placement vector to construct its device assignment.

Phase 4: Initial Fitness Evaluation. The penalty-aware fitness of each placement vector is evaluated using the function defined in Subsection 5.5. The placement having the lowest fitness establishes the initial best solution.

Phase 5: Metaheuristic Population Update. The evaluated population is refined using the selected population-update mechanism described in Subsection 5.6. Either GAUpdate or PSOUpdate modifies the current placement population using its respective update operators and returns the updated placement vectors to the common LCRA workflow for assignment reconstruction and fitness evaluation.

Phase 6: Reassignment, Fitness Evaluation, and Best-Solution Tracing. The updated population becomes the current population. PBDA reconstructs the device assignments. When uncovered devices or upgrade candidates remain, CRR repairs the placement and PBDA subsequently revalidates it. The penalty-aware fitness of every resulting placement is then recomputed. The lowest-fitness solution found so far is retained. Phases 5 and 6 are repeated until the predefined maximum number of search iterations is reached.

Phase 7: Final Validation and Solution Reporting. After the iterative search terminates, PBDA is applied to the best placement to reconstruct its assignment state. If uncovered devices or upgrade candidates remain, CRR repairs the placement, after which PBDA is reapplied and the final penalty-aware fitness is recomputed. The framework then returns the validated placement vector, device-assignment set, uncovered-device set, cloudlet resource usages, and final fitness value.

The complete workflow separates the problem-specific LCRA components from the selected metaheuristic population-update mechanism. Problem-aware initialization constructs the initial search population, PBDA converts each placement vector into device assignments, CRR handles residual coverage and resource infeasibility, and the penalty-aware fitness function provides a common basis for evaluating candidate solutions. GA and PSO differ only in how they generate updated placement populations, while the initialization, assignment, repair, evaluation, and final-validation mechanisms remain common. This modular organization enables both update strategies to be evaluated under the same LCRA processing conditions.

5.8. Computational Complexity and Scalability

Let N, M, and L denote the numbers of end devices, candidate points, and cloudlets, respectively. The following conservative bounds describe the supplied implementation, assuming constant-time numerical operations and expected constant-time dictionary and set access. Distance and static-latency matrices are treated as available inputs. The bounds concern operation counts, not measured execution times.

Component	**Conservative time bound**
LACI	$\mathcal{O}(LM[NM + N\log N + L + \log M])$
BCLI	$\mathcal{O}(NM + LM[N + \log M])$
CFI	$\mathcal{O}(LM[N + \log M])$
PBDA	$T_A = \mathcal{O}(M + NL + N\log N)$
CRR	$T_R = \mathcal{O}((R+1)NML[T_A + T_D])$
PopInit	$\mathcal{O}(A_{\text{total}}[T_S + T_A + T_R + M])$
Fitness evaluation after assignment and repair	$\mathcal{O}\big(N^2[L + M] + T_D\big)$

Here, R is the maximum number of coverage-repair repetitions, T_S is the largest cost of one seed-construction call, and $T_D = \mathcal{O}(N[M + L + N])$ bounds total-latency evaluation with the implemented state validation and finite-source queueing calculations. A_{total} counts attempts across the main, emergency, and final-completion stages and is bounded by $2\lambda_A N_{pop} + \max(F_{\min}, \lambda_A N_{pop}^2)$. The fitness bound applies to the assignment states constructed by PBDA, with at most one service association per device.

LACI includes device sorting and repeated latency-reference scans, whereas BCLI and CFI primarily examine cloudlet–point–device combinations. PBDA constructs reachable pairs and orders devices by assignment difficulty. CRR can be more expensive because radius-extension trials repeatedly invoke assignment and latency evaluation. PopInit multiplies these candidate-processing costs by its bounded generation attempts. These bounds explain why runtime depends on both instance dimensions and repair activity; they do not establish scalability beyond the evaluated instances.

6. Experimental Setup

This section describes the datasets, parameter settings, comparison methods, and evaluation metrics used to assess LCRA initialization. An extensive set of experiments are implemented in Python and executed on a MacBook Pro equipped with an Apple M3 Pro chip and 36 GB of RAM.

6.1. Parameters and Dataset Description

The experimental evaluation uses cloudlet-placement instances derived from the New York City Wi-Fi hotspot dataset adopted in the ACP study by Bhatta and Mashayekhy [7]. Five urban regions with different problem

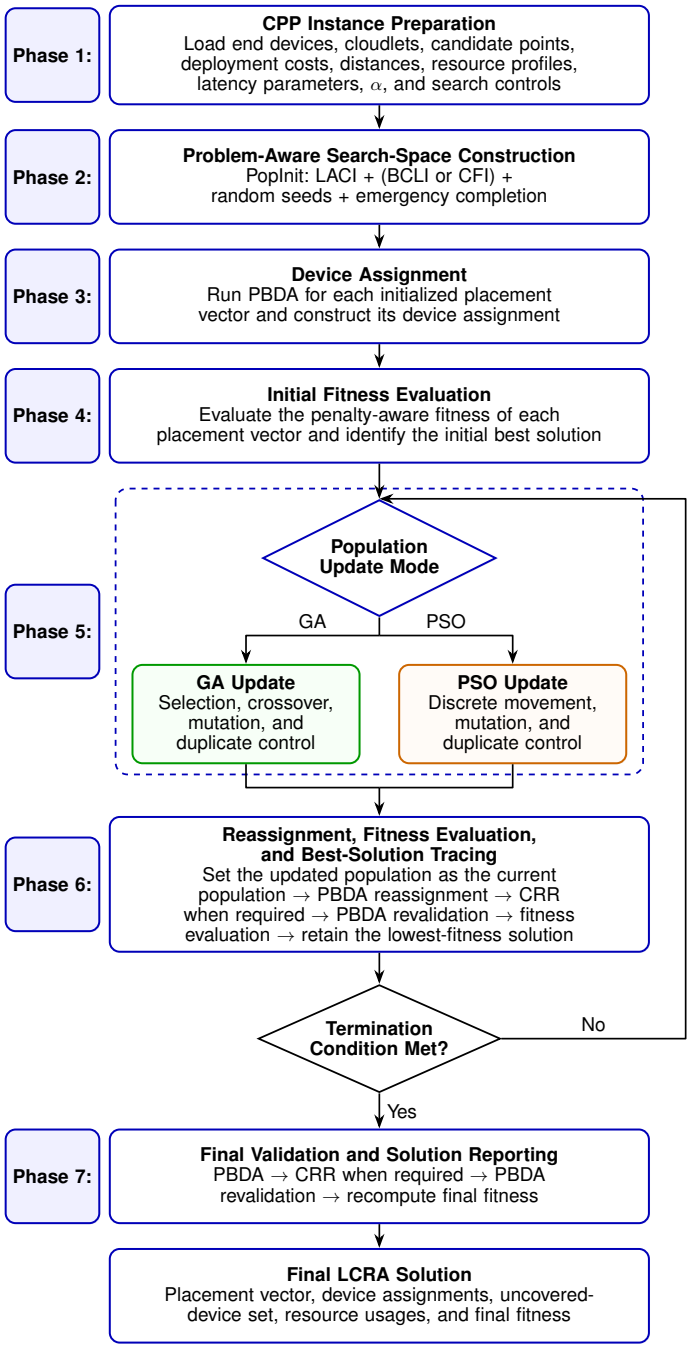


Figure 3. End-to-end execution workflow of the LCRA framework with selectable GA and PSO population-update modes.

scales are considered: Staten Island, Bronx, Queens, Brooklyn, and Manhattan. For each region, 30 sampled CPP instances are evaluated. Each instance contains its own candidate points, available cloudlets, end devices, deployment costs, and the information required to evaluate communication latency.

Each CPP instance is represented by five CSV files: `points.csv`, `device.csv`, `cloudlet.csv`, `costs.csv`, and `latencies.csv`. The first four files specify the candidate points, end devices, available

cloudlets, and cloudlet deployment costs, respectively. The `latencies.csv` file contains the original distance-based latency values associated with the ACP formulation.

The values stored in `latencies.csv` are retained only as part of the original ACP dataset and are not used during the execution or quantitative evaluation of LCRA. Instead, the latency values required by the proposed framework are computed on demand according to the latency model introduced in Section 3. The communication-related components are calculated from the corresponding end-device and candidate-point information, whereas the processing and queueing components are evaluated according to the current placement and device-assignment state. Consequently, the latency evaluation remains consistent with the system model throughout the optimization process.

Table 3 summarizes the ranges of candidate points, cloudlets, and end devices across the sampled instances.

Table 3. Characteristics of the sampled CPP instances

Borough	Candidate points	Cloudlets	End devices
Staten Island	7–10	5–8	59–71
Bronx	30–40	24–31	190–213
Queens	57–69	44–52	321–355
Brooklyn	81–97	61–73	428–451
Manhattan	108–127	81–96	1027–1091

The main system, initialization, repair, and search parameters are summarized in Table 4. These parameters are fixed across all evaluated datasets to provide a consistent experimental setting.

Table 4. Main experimental parameters

Parameter	Value
Task size τ_{e_i}	50 KB
Bandwidth B	5,000 KB/s
Signal propagation speed v_s	2×10^8 m/s
Instruction density D	1×10^6 cycles/KB
Arrival rate λ_j	0.1 tasks/s
Upgrade threshold τ_{up}	0.9
LACI budget factor B_f	1.0
LACI weights $(\gamma_1, \gamma_2, \gamma_3, \gamma_4)$	$(0.25, 0.45,$ $0.15, 0.15)$
Utilization target u_{tar}	0.65
Attempt coefficient λ_A	20
Minimum final-completion attempt budget $F_{\min}$	1000
Global experiment base seed	500

The experiments use fixed coefficients $\lambda_{\text{cov}} = \lambda_{\text{struct}} = \lambda_{\text{link}} = \lambda_{\text{cons}} = 3$, with $\lambda_{\text{cap},r} = 3$ for each resource dimension. This uniform setting is treated as a fixed experimental design choice, not an optimized parameter configuration. Its sensitivity was not evaluated in this study.

The GA uses tournament selection of size 3, uniform crossover with probability 0.8, mutation with probability 0.15, and approximately 5% elitism with at least one elite.

The discrete PSO uses current-particle, personal-best, and global-best influences of 0.4, 0.3, and 0.3, respectively, followed by mutation with probability 0.1.

Both population updaters use a duplicate-child attempt limit of 200.

For each borough, sample index $m \in \{1, \ldots, 30\}$ is assigned the instance-specific random seed $s_0 = 500 + m$, giving seeds 501–530. The same instance-seed mapping is used across the compared methods and preference settings to provide consistent, reproducible experimental conditions.

The preference parameter is evaluated at $\alpha \in \{0.1, 0.5, 0.9\}$. These values represent the latency-focused, balanced, and cost-focused optimization settings, respectively.

The population size is adjusted according to the scale of each dataset, as shown in Table 5. Smaller populations are assigned to larger problem instances to maintain practical computational requirements while preserving the same experimental procedure across all boroughs. Since each borough contains 30 sampled instances, the total number of initialized candidates for one method at a given α value is $30N_{pop}$.

Table 5. Dataset-specific population sizes

Borough	N_{pop} per instance
Staten Island	50
Bronx	50
Queens	40
Brooklyn	30
Manhattan	20

PopInit constructs the initial population from coverage-oriented, balanced or cost-oriented, and random candidates. The proportion assigned to each component depends on the selected α regime. Table 6 summarizes the corresponding initialization quotas and acceptance thresholds. The emergency-completion threshold is intentionally lower than the standard initialization threshold, such that $\tau_{emg} < \tau_{init}$, allowing additional candidates to be accepted when the required population size cannot otherwise be reached.

Table 6. α-dependent PopInit quotas and acceptance thresholds

Regime	N_{cov}	N_{bla}	τ_{init}	τ_{emg}
$\alpha \leq 0.3$	$\lfloor 0.50N_{pop} \rfloor$	$\lfloor 0.20N_{pop} \rfloor$	0.80	0.60
$0.3 < \alpha < 0.8$	$\lfloor 0.35N_{pop} \rfloor$	$\lfloor 0.25N_{pop} \rfloor$	0.75	0.55
$\alpha \geq 0.8$	$\lfloor 0.25N_{pop} \rfloor$	$\lfloor 0.45N_{pop} \rfloor$	0.70	0.50

The three preference regimes remain distinct: $0 \leq \alpha \leq 0.3$ is latency-focused, $0.3 < \alpha < 0.8$ is balanced, and $0.8 \leq \alpha \leq 1$ is cost-focused. PopInit combines LACI, BCLI, and random candidates in the first two regimes, with different quotas. In the cost-focused regime, CFI replaces BCLI in the structured component. These mixtures provide different starting solutions without changing the selected cost–latency preference. The entries in Table 6 are target quotas; rounding, unsuccessful attempts, and completion procedures can affect the final composition.

6.2. Baseline and Comparative Algorithms

LCRA-GA is compared with Random-GA, and LCRA-PSO with Random-PSO, to isolate the effect of LCRA initialization while keeping the corresponding population-update mechanism unchanged. LMACP provides an additional external baseline.

Approximate Cloudlet Placement (ACP), introduced by Bhatta and Mashayekhy [7], is the original method from which LMACP is adapted. Under ACP's original assumptions, its paper proves approximation factors of two for latency and four for deployment cost. These guarantees cannot automatically be claimed for the modified latency model without further proof. ACP's original distance-based results are therefore excluded from the time-based quantitative comparisons.

Latency-Modified Approximate Cloudlet Placement (LMACP) retains the supplied ACP placement procedure but replaces its original distance-based latency calculation with a four-component latency model. Its LP minimizes deployment cost, while communication latency, calculated using Equation (12), guides filtering and rounding. After cloudlet placement and device assignment are established, total latency is recomputed using Equation (11), incorporating propagation, transmission, processing, and queueing delays. LMACP and LCRA address the same underlying CPP using different solution procedures.

For the GA-based comparison, Random-GA and LCRA-GA employ the same GA population-update mechanism and differ only in the construction of the initial population. Random-GA starts from a fully random population, whereas LCRA-GA uses the problem-aware PopInit procedure. Their comparison therefore measures the effect of LCRA initialization when GA is used as the population-update mechanism.

Similarly, Random-PSO and LCRA-PSO use the same discrete PSO population-update mechanism. Random-PSO begins with a randomly generated population, whereas LCRA-PSO uses the problem-aware PopInit procedure. This comparison evaluates whether the benefit of the proposed initialization strategy is maintained when PSO is used instead of GA.

6.3. Performance Metrics

The experimental evaluation assesses both placement quality and computational efficiency using three performance metrics: deployment cost, total latency, and runtime.

Total deployment cost represents the sum of the deployment costs of all cloudlets selected in a solution and is expressed in V_u. Lower values indicate more cost-efficient placement decisions.

Total latency is defined as the sum of the service latencies associated with all device–cloudlet assignments. It is calculated using the four-component latency model introduced in Section 3, which includes propagation, transmission, processing, and queueing delays. The latency is computed and reported in seconds. Lower total latency indicates better service responsiveness.

Runtime is taken from the experimental results and reported in milliseconds. It is used to compare the computational time of the LCRA and random-initialized variants.

Final feasibility validation. A run is counted as successful only when the final total residual-violation penalty is zero. All residual terms are nonnegative and their penalty coefficients are strictly positive. Therefore, zero total penalty requires every implemented residual term to be zero. The success rate is the number of successful runs divided by the total number of evaluated runs. Coverage and service-link checks use the repaired candidate's effective communication radii, as described in Section 5.3. Table 7 summarizes the constraint groups and their corresponding final validation checks.

Table 7. Implemented final feasibility checks.

Constraint group	Final validation check
Coverage	No end device remains unassigned.
Resource capacity	Assigned processor, memory, and storage demands do not exceed deployed cloudlet capacities.
Deployment structure	Each point hosts at most one cloudlet, and each cloudlet is deployed at most once.
Service links	Each assignment refers to the deployed cloudlet and satisfies its effective communication range.
Assignment consistency	Cloudlet-level, point-level, and detailed service assignments agree, with one service association per device.

Deployment cost and total latency are evaluated separately for each value of α and averaged over the 30 sampled instances associated with each urban region. Runtime results are aggregated over the 30 instances and the three evaluated preference settings, $\alpha \in \{0.1, 0.5, 0.9\}$.

6.4. Statistical Analysis

For each borough, preference setting, optimizer, and metric, LCRA and Random results are paired by sampled-instance identity, giving 30 matched pairs per comparison. We report means and sample standard deviations, mean paired differences, pointwise 95% bias-corrected and accelerated bootstrap confidence intervals using 20,000

paired resamples, and Cohen's paired effect size d_z. The latter is calculated as the mean paired difference divided by the sample standard deviation of the paired differences.

Differences are defined as Random minus LCRA. An exact two-sided paired sign test assesses whether positive and negative differences are equally likely, excluding zero differences. This common test avoids normality and symmetry assumptions. Holm correction controls the family-wise error rate at 5% across all 90 comparisons. Sampled instances within each comparison are treated as independent; observations are not pooled across settings. The confidence intervals and d_z describe difference magnitudes, whereas the sign tests assess directional consistency. Variability is assessed across sampled instances, not across repeated optimizer seeds.

7. Results and Discussion

The aim of this section is to study the Cloudlet Placement Problem using the LCRA framework developed in the preceding sections. The experimental study assesses the impact of LCRA initialization on deployment cost, total latency, and runtime. For this purpose, both GA and PSO are first executed using the proposed LCRA initialization, resulting in the LCRA-GA and LCRA-PSO variants. The same GA and PSO update mechanisms are then executed under purely random initialization, producing the corresponding Random-GA and Random-PSO variants. This paired experimental design isolates the effect of the initialization strategy while keeping the underlying metaheuristic update mechanism unchanged. LMACP is additionally included as an external approximation-based baseline for comparative evaluation.

Final feasibility results. Each of the 60 submitted result files reports 30 successful runs out of 30, corresponding to 1,800 runs across the four GA/PSO variants, five boroughs, and three preference settings. Thus, all reported runs passed the implemented final feasibility checks, with complete device coverage and zero residual penalties under the repair-aware evaluation.

Paired statistical results. The paired comparisons show consistent improvements in both deployment cost and average total latency for LCRA relative to the corresponding random-initialized GA and PSO baselines. Across the 90 comparisons, every group contained 30 differences with the same sign: LCRA had lower deployment cost and total latency, but higher runtime. Each exact two-sided sign test gave $p = 1.863 \times 10^{-9}$ and a Holm-adjusted p-value of 1.676×10^{-7}. Table 8 summarizes the main cost-focused and latency-focused comparisons.

7.1. Deployment Cost

To evaluate the deployment-cost performance of the proposed LCRA framework, the experiments adopt the cost-focused setting, defined by $0.8 \leq \alpha \leq 1$. In this regime, greater emphasis is assigned to deployment-cost minimization. Accordingly, $\alpha = 0.9$ is selected for the cost-focused evaluation.

Using this setting, LCRA-GA and LCRA-PSO are evaluated against their corresponding random-initialization variants, Random-GA and Random-PSO, across Staten Island, Bronx, Queens, Brooklyn, and Manhattan. For each method and urban region, the deployment cost is averaged over 30 sampled CPP instances and expressed in the user-defined virtual cost unit V_u.

Fig. 4 shows the same ordering in all five boroughs: LCRA-GA achieves the lowest average deployment cost, followed by LCRA-PSO, Random-GA, Random-PSO, and LMACP. Because each LCRA/random pair uses the same population-update mechanism and evaluation pipeline, the lower costs obtained by LCRA-GA and LCRA-PSO demonstrate the contribution of problem-aware initialization. Starting from placements informed by cost, latency, coverage, and resource conditions gives GA and PSO economically favorable structures to refine instead of arbitrary deployments. Although absolute cost increases from Staten Island to Manhattan as the problem scale grows, LCRA retains its advantage across all instance sizes.

Table 9 quantifies these reductions. Relative to random initialization, LCRA-GA reduces cost by 9.53–12.60%, while LCRA-PSO reduces it by 9.19–12.42%. Compared with the evaluated LMACP implementation, the reported

Table 8. Paired statistical summary of the main cost-focused and latency-focused comparisons.

Deployment cost at $\alpha = 0.9$ (all values in V_u)					
Updater	**Borough**	**LCRA mean ± SD**	**Random mean ± SD**	**Mean reduction [95% CI]**	d_z
GA	Staten Island	1914.60 ± 231.97	2190.72 ± 267.76	276.12 [263.08, 291.31]	6.88
GA	Bronx	6360.00 ± 328.55	7211.14 ± 366.95	851.14 [828.81, 874.51]	12.93
GA	Queens	10447.60 ± 394.03	11790.28 ± 474.96	1342.68 [1292.69, 1389.52]	9.65
GA	Brooklyn	13642.78 ± 567.66	15276.84 ± 612.61	1634.06 [1588.87, 1685.44]	12.05
GA	Manhattan	30901.93 ± 654.17	34156.23 ± 685.98	3254.29 [3162.42, 3348.18]	12.45
PSO	Staten Island	1939.26 ± 191.71	2214.18 ± 226.35	274.93 [262.15, 290.06]	6.93
PSO	Bronx	6510.00 ± 269.97	7356.33 ± 313.60	846.33 [820.40, 873.12]	11.46
PSO	Queens	10775.69 ± 449.72	12068.23 ± 507.38	1292.54 [1250.84, 1336.36]	10.59
PSO	Brooklyn	13992.78 ± 511.61	15614.27 ± 603.64	1621.49 [1558.88, 1682.79]	9.32
PSO	Manhattan	31796.07 ± 713.29	35011.96 ± 840.82	3215.89 [3117.51, 3315.29]	11.41
Average total latency at $\alpha = 0.1$ (means/SDs in s; reductions/CIs in ms)					
Updater	**Borough**	**LCRA mean ± SD (s)**	**Random mean ± SD (s)**	**Mean reduction [95% CI] (ms)**	d_z
GA	Staten Island	0.6556 ± 0.0326	0.6571 ± 0.0326	1.444 [1.423, 1.467]	22.60
GA	Bronx	2.0654 ± 0.0597	2.0716 ± 0.0596	6.163 [6.064, 6.267]	21.44
GA	Queens	3.4841 ± 0.0815	3.4963 ± 0.0814	12.170 [11.962, 12.353]	21.93
GA	Brooklyn	4.5466 ± 0.0630	4.5616 ± 0.0630	15.003 [14.752, 15.248]	21.13
GA	Manhattan	10.6511 ± 0.1332	10.6863 ± 0.1332	35.198 [34.588, 35.801]	20.62
PSO	Staten Island	0.6556 ± 0.0326	0.6571 ± 0.0327	1.483 [1.460, 1.507]	21.92
PSO	Bronx	2.0660 ± 0.0550	2.0724 ± 0.0551	6.424 [6.323, 6.525]	22.66
PSO	Queens	3.4880 ± 0.0812	3.5005 ± 0.0810	12.540 [12.305, 12.760]	19.53
PSO	Brooklyn	4.5528 ± 0.0628	4.5681 ± 0.0628	15.348 [15.094, 15.621]	20.61
PSO	Manhattan	10.6632 ± 0.1328	10.6995 ± 0.1331	36.290 [35.732, 36.864]	22.40

Each row uses 30 matched instances. Reductions are calculated as Random minus LCRA. Brackets contain pointwise 95% confidence intervals for the mean paired reduction.

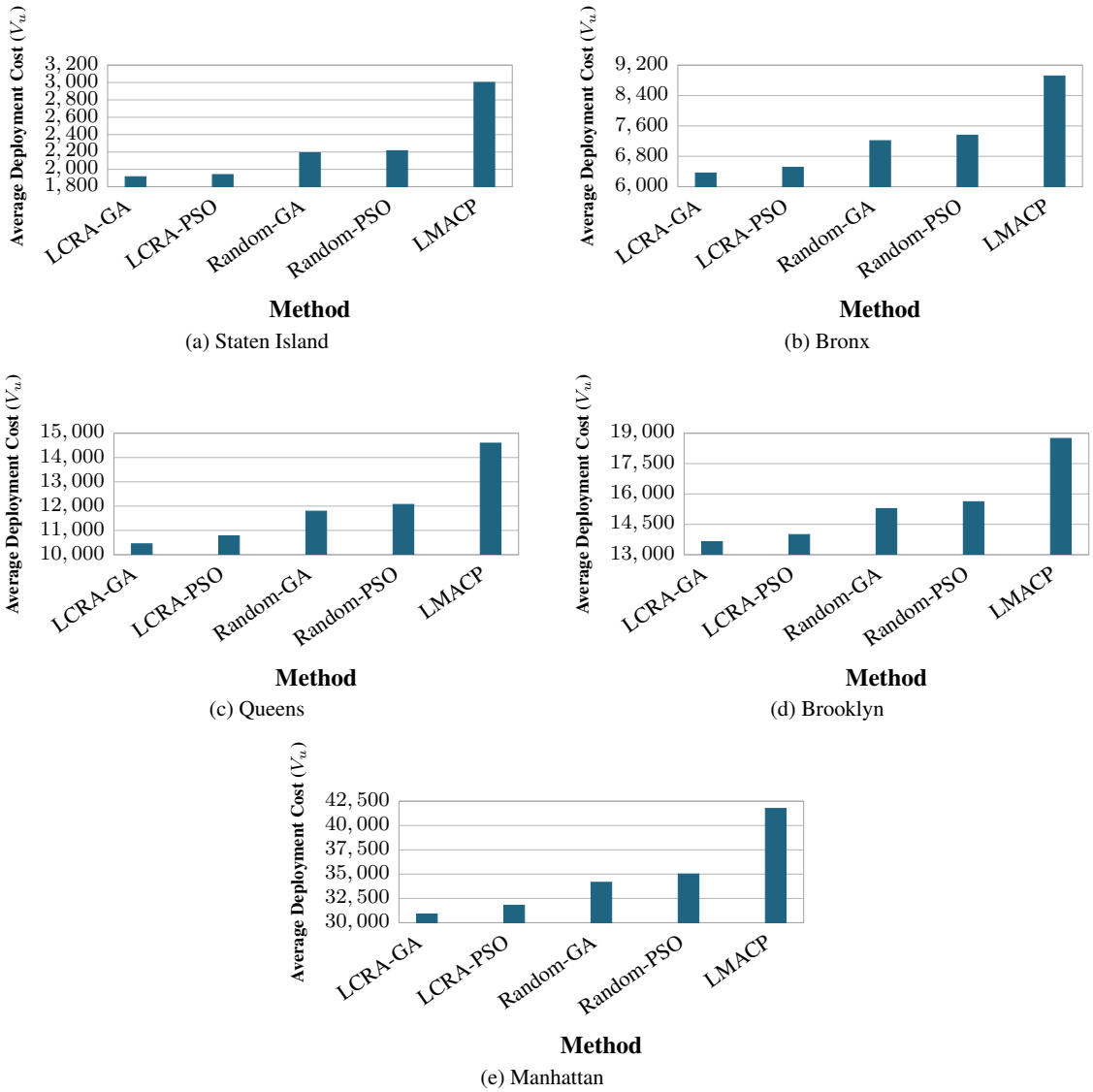
Table 9. Cost-improvement percentage of LCRA-based methods under the cost-focused setting $\alpha = 0.9$.

Borough	**LCRA-GA vs. Random-GA (%)**	**LCRA-PSO vs. Random-PSO (%)**	**LCRA-GA vs. LMACP (%)**	**LCRA-PSO vs. LMACP (%)**
Staten Island	12.60	12.42	36.21	35.39
Bronx	11.80	11.50	28.66	26.98
Queens	11.39	10.71	28.39	26.15
Brooklyn	10.70	10.38	27.20	25.34
Manhattan	9.53	9.19	25.99	23.85

deployment-cost values are lower by 25.99–36.21% for LCRA-GA and 23.85–35.39% for LCRA-PSO. The GA-based variant achieves the larger direct cost reduction in every borough. Although the percentage advantage gradually narrows as the instance scale increases, it remains positive for both update modes across all datasets.

Fig. 5 shows that the average deployment cost in Manhattan decreases for all GA/PSO variants as α increases from 0.1 to 0.9. This behavior is consistent with the scalarized objective, which places greater emphasis on cost at larger α. LCRA-GA and LCRA-PSO remain below their random-initialized counterparts at every tested value, showing that the improvement is attributable not only to the objective preference but also to the initial search space.

The best-across-α results in Table 10 confirm that LCRA-GA achieves the lowest average deployment cost in every borough. Its best setting is $\alpha = 0.5$ for Staten Island and $\alpha = 0.9$ for the other four boroughs. These findings indicate that informed initial placements allow crossover and mutation to refine economically favorable



(a) Staten Island (b) Bronx

(c) Queens (d) Brooklyn

(e) Manhattan

Figure 4. Average deployment cost at $\alpha = 0.9$ across the five boroughs. Borough-specific y-axis ranges are used.

Table 10. Best average deployment cost (V_u) by borough across the evaluated methods.

Borough	**LMACP**	**LCRA-GA**	**Random-GA**	**LCRA-PSO**	**Random-PSO**	**Best method**
Staten Island	3001	**1881**	2171	1905	2192	LCRA-GA ($\alpha = 0.5$)
Bronx	8915	**6360**	7211	6510	7356	LCRA-GA ($\alpha = 0.9$)
Queens	14590	**10448**	11790	10776	12068	LCRA-GA ($\alpha = 0.9$)
Brooklyn	18741	**13643**	15277	13993	15614	LCRA-GA ($\alpha = 0.9$)
Manhattan	41756	**30902**	34156	31796	35012	LCRA-GA ($\alpha = 0.9$)

structures rather than begin from arbitrary deployments. Together with PBDA and the penalty-aware evaluation, this initialization enables both update modes to reduce cost without favoring infeasible placements.

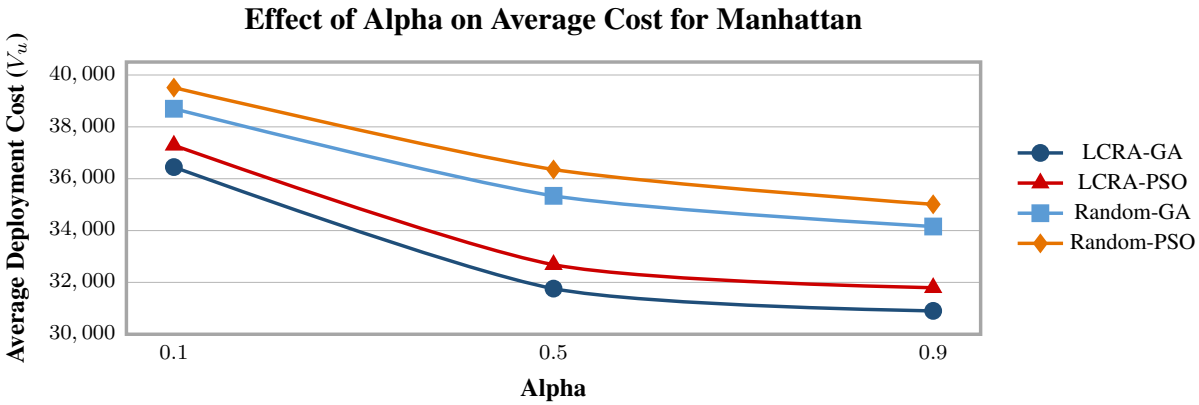


Figure 5. Effect of the preference parameter α on the average deployment cost for Manhattan.

Table 11. Latency-improvement percentage of LCRA-based methods under the latency-focused setting $\alpha = 0.1$.

Borough	**LCRA-GA vs. Random-GA (%)**	**LCRA-PSO vs. Random-PSO (%)**	**LCRA-GA vs. LMACP (%)**	**LCRA-PSO vs. LMACP (%)**
Staten Island	0.22	0.23	1.06	1.06
Bronx	0.30	0.31	0.99	0.96
Queens	0.35	0.36	1.81	1.70
Brooklyn	0.33	0.34	1.82	1.69
Manhattan	0.33	0.34	1.82	1.71

7.2. Latency Performance

Latency is calculated using the four-component model in Section 3 and reported in seconds. Each value is averaged over 30 sampled CPP instances. LMACP and all four GA/PSO variants use this common time-based model. ACP is excluded from the quantitative latency comparison because its original latency criterion is distance-based, but it remains the source method from which LMACP is derived.

Under the latency-focused setting $\alpha = 0.1$, Fig. 6 shows that LCRA-GA outperforms Random-GA and that LCRA-PSO outperforms Random-PSO in every borough. Both LCRA variants also obtain lower average total latency than LMACP. Absolute latency increases with instance scale, but the relative ordering remains stable. Because the corresponding LCRA and random variants use identical update and evaluation mechanisms, these results demonstrate the contribution of problem-aware initialization under both update modes. LCRA provides placements informed by coverage, latency, cost, and resource conditions, while PBDA maps them to reachable, capacity-feasible assignments and favors lower communication delay. This interaction gives GA and PSO feasible, delay-aware structures to refine instead of requiring them to correct weak random placements.

Table 11 shows that LCRA-GA reduces latency relative to Random-GA by 0.22–0.35%, whereas LCRA-PSO reduces it relative to Random-PSO by 0.23–0.36%. The PSO-based variant provides a slightly larger direct initialization improvement in each borough, although the difference is small. Compared with the evaluated LMACP implementation, the reported latency values are lower by 0.99–1.82% for LCRA-GA and 0.96–1.71% for LCRA-PSO. The paired results support LCRA's contribution to improving both deployment cost and average total latency relative to the corresponding random-initialized GA and PSO baselines. These improvements remain statistically significant after Holm correction. At $\alpha = 0.1$, the reduction in average total latency ranges from 1.444 to 35.198 ms for GA and from 1.483 to 36.290 ms for PSO. These values are averaged across the 30 sampled instances and do not represent per-device delay reductions. The large d_z values indicate consistent paired improvements; d_z is a standardized measure, not a percentage reduction. The absolute and percentage gains, together with deployment-cost savings and additional runtime, provide the basis for assessing the practical trade-off.

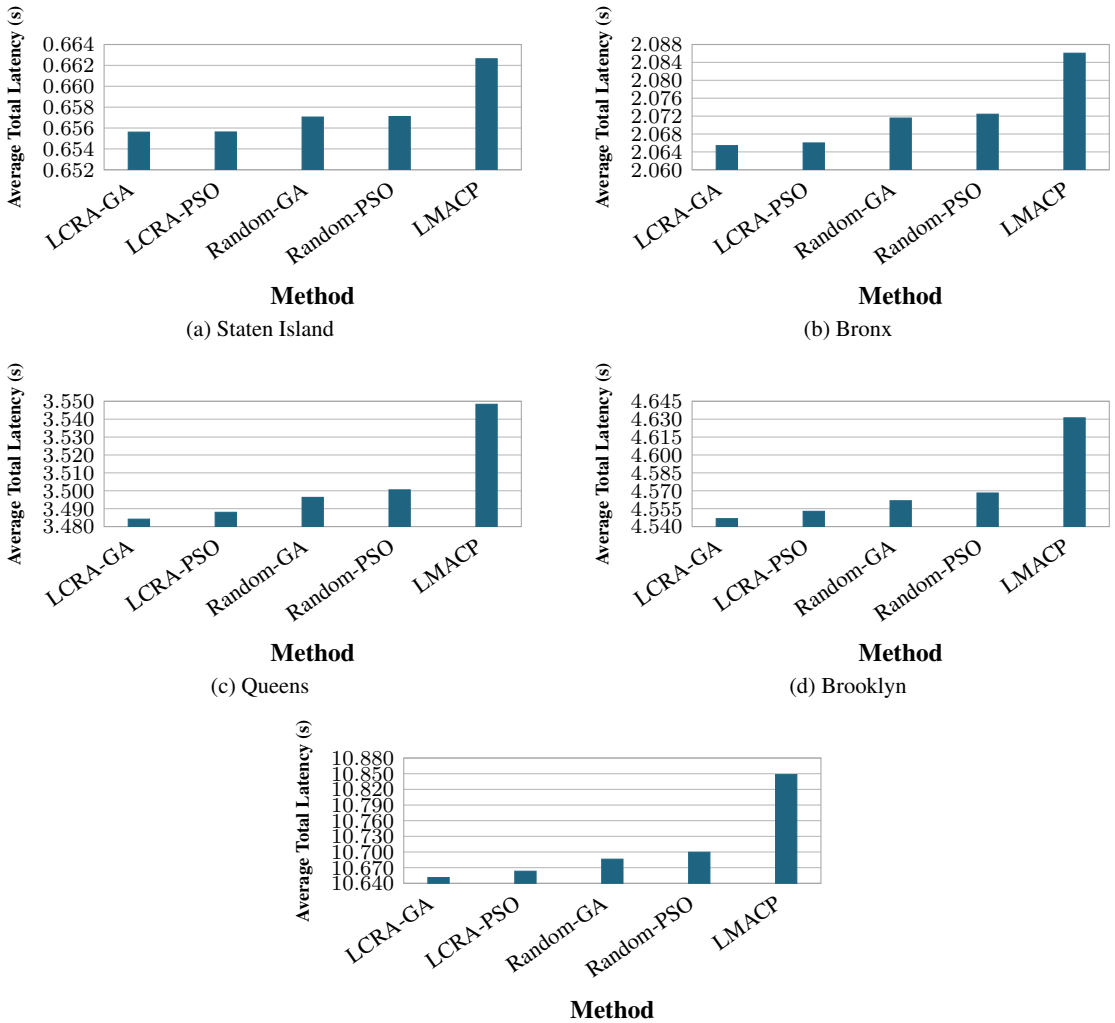


Figure 6. Average total latency at $\alpha = 0.1$ across the five boroughs. Borough-specific y-axis ranges are used.

As shown in Fig. 7, Manhattan's average total latency increases as α changes from 0.1 to 0.9 because the objective progressively shifts toward cost minimization. Nevertheless, both LCRA variants retain lower latency than their random counterparts at every tested value. Together with Fig. 5, this result confirms the intended preference behavior: increasing α lowers deployment cost while increasing latency.

Table 12 identifies LCRA-GA at $\alpha = 0.1$ as the lowest-latency method in every borough. Thus, LCRA improves both GA and PSO relative to random initialization, while the GA update achieves the best overall cost and latency results across the evaluated methods. The reported success rates indicate that these latency results passed the implemented final checks for coverage, resource capacity, deployment structure, service links, and assignment consistency.

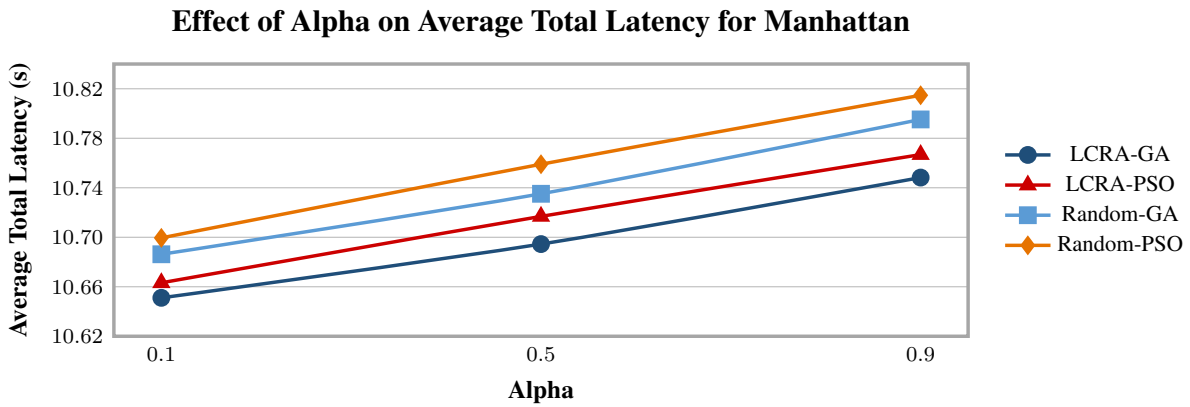


Figure 7. Effect of the preference parameter α on the average total latency for Manhattan.

Table 12. Best average total latency, in seconds, by borough across the evaluated methods.

Borough	LMACP	LCRA-GA	Random-GA	LCRA-PSO	Random-PSO	Best method
Staten Island	0.663	**0.656**	0.657	0.656	0.657	LCRA-GA ($\alpha = 0.1$)
Bronx	2.086	**2.065**	2.072	2.066	2.072	LCRA-GA ($\alpha = 0.1$)
Queens	3.548	**3.484**	3.496	3.488	3.501	LCRA-GA ($\alpha = 0.1$)
Brooklyn	4.631	**4.547**	4.562	4.553	4.568	LCRA-GA ($\alpha = 0.1$)
Manhattan	10.848	**10.651**	10.686	10.663	10.699	LCRA-GA ($\alpha = 0.1$)

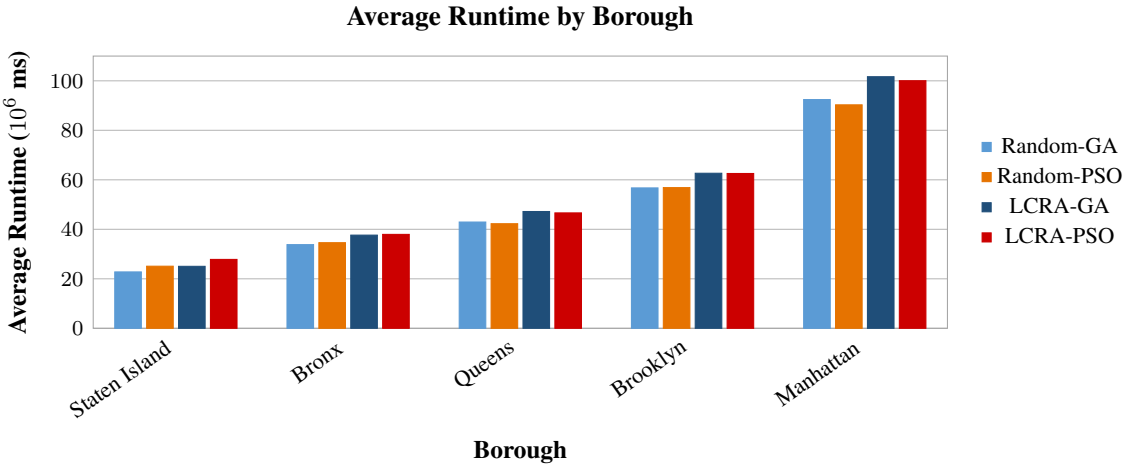


Figure 8. Average runtime by borough for LCRA-GA, Random-GA, LCRA-PSO, and Random-PSO. Runtime values are averaged over $\alpha = 0.1$, 0.5, and 0.9.

7.3. Runtime Performance

Runtime evaluates the computational cost of replacing random initialization with problem-aware population construction. For each borough and method, Fig. 8 reports the average across the 30 sampled instances and the three evaluated settings, $\alpha = 0.1$, 0.5, and 0.9.

Figure 8 shows that Random-GA and Random-PSO are faster than their corresponding LCRA variants. This difference may reflect the additional computational effort associated with structured population construction and

Table 13. Average runtime and LCRA overhead relative to the random-initialized baselines.

Borough	Random-GA (ms)	LCRA-GA (ms)	Overhead (%)	Random-PSO (ms)	LCRA-PSO (ms)	Overhead (%)
Staten Island	22,783,412	25,045,406	9.93	25,094,540	27,838,279	10.93
Bronx	33,812,436	37,650,000	11.35	34,593,933	37,950,000	9.70
Queens	42,939,246	47,191,803	9.90	42,262,487	46,631,760	10.34
Brooklyn	56,729,853	62,654,032	10.44	56,850,456	62,557,504	10.04
Manhattan	92,428,977	101,663,596	9.99	90,297,088	100,042,691	10.79

Runtime values are averaged over the 30 sampled instances and three preference settings and rounded to the nearest millisecond. Overheads are calculated before rounding.

emergency completion when necessary. Runtime also increases from Staten Island to Manhattan as the numbers of devices, cloudlets, and candidate points grow. The nonuniform runtime difference between LCRA-GA and LCRA-PSO across the boroughs indicates that execution time depends on both problem scale and the selected population-update mechanism.

Table 13 shows that LCRA introduces average runtime overheads of 9.90–11.35% for GA and 9.70–10.93% for PSO. Together with the preceding cost and latency results, these values identify the practical trade-off: LCRA improves both deployment cost and average total latency relative to the corresponding random-initialized variants, while requiring additional computation. It is therefore a candidate for offline planning or periodic placement updates when the available computation window accommodates the reported runtime and placement quality is the priority. For time-critical decisions, suitability depends on the application's deadline; the reported averages alone do not establish deadline compliance or real-time performance.

8. Conclusion

This paper proposed the Latency-Cost-Requirement-Aware (LCRA) framework for improving the population initialization of metaheuristics applied to the heterogeneous Cloudlet Placement Problem. LCRA replaces purely random initialization with latency-oriented, balanced cost–latency, cost-focused, and diversity-preserving seeding strategies. The generated placements are then processed through Priority-Based Device Assignment, Coverage and Resource Repair, and penalty-aware fitness evaluation before refinement by GA or PSO. The study also introduced LMACP as a time-compatible approximation-based baseline by adapting ACP to the four-component latency model used in this work.

The experimental results show that problem-aware initialization improves both GA and PSO. Compared with the corresponding random-initialization variants, LCRA reduced average deployment cost by 9.19–12.60% under the cost-focused setting and average total latency by 0.22–0.36% under the latency-focused setting. Compared with the evaluated LMACP implementation, the reported deployment-cost and average total latency values were lower by up to 36.21% and 1.82%, respectively. LCRA-GA achieved the lowest average deployment cost and latency across all evaluated urban regions. All evaluated metaheuristic runs passed the implemented final feasibility checks, indicating complete device coverage and zero residual constraint violations under the repair-aware evaluation. These improvements were obtained with runtime overheads of 9.90–11.35% for GA and 9.70–10.93% for PSO.

Within the evaluated NYC-derived static instances, the results demonstrate that incorporating problem-specific knowledge into the initial search space can improve cloudlet-placement quality while preserving feasibility and maintaining a moderate computational overhead. The empirical conclusions concern the tested GA/PSO configurations; broader geographic, workload, and optimizer generalization requires separate evaluation.

9. Future Work

Future work will investigate finer-grained preference settings, particularly near $\alpha = 0.3$ and $\alpha = 0.8$, to assess their effects on solution quality and runtime.

Future work will investigate exact solutions for small instances under the scalarized objective, resource constraints, and radius-extension cost rules used in this study. These benchmarks would support valid lower bounds and optimality-gap assessment.

Further work will examine periodically updated placements under changing device locations and workloads, including reassignment and migration costs. The common placement-vector and evaluation structure also provides a basis for investigating a discrete differential-evolution updater and compatible heuristic or exact-solver seeds. Adaptive seeding could adjust strategy quotas according to observed search progress. These directions require dedicated implementation and evaluation.

Availability and Reproducibility

Code availability. The Python implementations of LCRA and the LMACP baseline are available from the corresponding author upon reasonable request.

Data availability. The processed CPP instances described in Section 6.1 and the experimental result files underlying the reported comparisons are available from the corresponding author upon reasonable request.

Reproducibility. The experimental input structure, parameter settings, population sizes, and instance-specific seed mapping are documented in Section 6.1. Sections 6.2 and 6.3 describe the comparison procedures, evaluation metrics, and final feasibility checks.

REFERENCES

1. Saeid Jahandar, Ibraheem Shayea, Emre Gures, Ayman A. El-Saleh, Mustafa Ergen, and Mohammad Alnakhli. Handover decision with multi-access edge computing in 6g networks: A survey. *Results in Engineering*, 25:103934, 2025.
2. Soule Issa Loutfi, Ibraheem Shayea, Ufuk Tureli, Ayman A. El-Saleh, and Waheeb Tashan. An overview of mobility awareness with mobile edge computing over 6g network: Challenges and future research directions. *Results in Engineering*, 23:102601, July 2024.
3. Shanshan Zhang, Jiong Yu, and Mingjian Hu. An edge server placement based on graph clustering in mobile edge computing. *Scientific Reports*, 14:29986, 2024.
4. Xiaojian Zhu and MengChu Zhou. *Multi-objective Optimization Approach to High-Performance Cloudlet Deployment and Task Offloading in Mobile-Edge Computing*, pages 89–119. Internet of Things. Springer, Cham, Switzerland, 2024.
5. Song Yang, Fan Li, Meng Shen, Xu Chen, Xiaoming Fu, and Yu Wang. Cloudlet placement and task allocation in mobile edge computing. *IEEE Internet of Things Journal*, 6(3):5853–5863, 2019.
6. Jianshan Zhang, Ming Li, Xianghan Zheng, and Ching-Hsien Hsu. A time-driven cloudlet placement strategy for workflow applications in wireless metropolitan area networks. *Sensors*, 22(9):3422, 2022.
7. Dixit Bhatta and Lena Mashayekhy. A bifactor approximation algorithm for cloudlet placement in edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1787–1798, 2022.
8. Balázs Sonkoly, János Czentye, Márk Szalay, Balázs Németh, and László Toka. Survey on placement methods in the edge and beyond. *IEEE Communications Surveys & Tutorials*, 23(4):2590–2629, 2021.
9. Ali Asghari and Mohammad Karim Sohrabi. Server placement in mobile cloud computing: A comprehensive survey for edge computing, fog computing and cloudlet. *Computer Science Review*, 51:100616, 2024.
10. Yanling Shao, Zhen Shen, Siliang Gong, and Hanyao Huang. Cost-aware placement optimization of edge servers for iot services in wireless metropolitan area networks. *Wireless Communications and Mobile Computing*, 2022:8936576, 2022.
11. Layth Muwafaq, Nor K. Noordin, Mohamed Othman, Alyani Ismail, and Fazirulhisyam Hashim. A survey on cloudlet computation optimization in the mobile edge computing environment. *International Journal of Advanced Computer Science and Applications*, 14(1):520–532, 2023.
12. Dixit Bhatta and Lena Mashayekhy. Generalized cost-aware cloudlet placement for vehicular edge computing systems. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 159–166, 2019.
13. Tayebeh Bahreini and Daniel Grosu. Efficient algorithms for multi-component application placement in mobile edge computing. *IEEE Transactions on Cloud Computing*, 10(4):2550–2563, 2022.
14. Zichuan Xu, Lizhen Zhou, Sid Chi-Kin Chau, Weifa Liang, Haipeng Dai, Lixing Chen, Wenzheng Xu, Qiufen Xia, and Pan Zhou. Near-optimal and collaborative service caching in mobile edge clouds. *IEEE Transactions on Mobile Computing*, 22(7):4070–4085, 2023.
15. Yanpei Liu, Yanru Bin, Ningning Chen, and Shuaijie Zhu. Caching placement optimization strategy based on comprehensive utility in edge computing. *Applied Sciences*, 13(16):9229, 2023.

16. Samuel Rac and M. Brorsson. Cost-aware service placement and scheduling in the edge-cloud continuum. *ACM Transactions on Architecture and Code Optimization*, 21(2):29:1–29:24, 2024.
17. Lei Yang, Jiannong Cao, Guanqing Liang, and Xu Han. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers*, 65(5):1440–1452, 2016.
18. Yuncan Zhang, Weifa Liang, Wenzheng Xu, Zichuan Xu, and Xiaohua Jia. Cost minimization of digital twin placements in mobile edge computing. *ACM Transactions on Sensor Networks*, 20(3):74:1–74:26, 2024.
19. Yuncan Zhang, Weifa Liang, Zichuan Xu, and Xiaohua Jia. Mobility-aware service provisioning in edge computing via digital twin replica placements. *IEEE Transactions on Mobile Computing*, 23(12):11295–11311, 2024.
20. Xuan Ai, Weifa Liang, Yuncan Zhang, and Wenzheng Xu. Fidelity-aware inference services in dt-assisted edge computing via service model retraining. *IEEE Transactions on Services Computing*, 18(4):2089–2102, 2025.
21. Adyson M. Maia, Yacine Ghamri-Doudane, Dario Vieira, and Miguel Franklin de Castro. An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in edge computing. *Computer Networks*, 194:108146, 2021.
22. Mohammadsadeq Garshasbi Herabad, Javid Taheri, Bestoun S. Ahmed, and Calin Curescu. E-psoga: An enhanced hybrid metaheuristic for optimal edge-to-cloud placement of services with multi-version components. *IEEE Access*, 13:151170–151188, 2025.
23. Mohamed Elkawkagy, Ibrahim A. Elgendy, Samia Allaoua Chelloug, and Heba Elbeh. Genetic algorithm-driven joint optimization of task offloading and resource allocation for fairness-aware latency minimization in mobile edge computing. *IEEE Access*, 13:118237–118248, 2025.
24. Rashmi Keshri and Deo Prakash Vidyarthi. An ml-based task clustering and placement using hybrid jaya–gray wolf optimization in fog-cloud ecosystem. *Concurrency and Computation: Practice and Experience*, 36(14):e8109, 2024.
25. Muralidhar Kurni, Ramesh Krishnamaneni, and Ashwin Narasimha Murthy. Osprey–lyrebird optimization-based resource allocation with optimal edge-server placement and offloading in mobile-edge server computing. *International Journal of Communication Systems*, 38(13):e70214, 2025.
26. Weili Si, Zhifeng Zhang, and Bo Wang. Edge server placement by a novel hybrid meta-heuristic algorithm with alternating iteration. *Digital*, 6(2):44, 2026.
27. Xiaodong Xing, Zhifeng Zhang, and Bo Wang. A novel meta-heuristic edge server placement algorithm for improving service quality. *Computers*, 15(5):324, 2026.
28. Fei Luo, Shuai Zheng, Weichao Ding, Joel Fuentes, and Yong Li. An edge server placement method based on reinforcement learning. *Entropy*, 24(3):317, 2022.
29. Mumraiz Khan Kasi, Sarah Abu Ghazalah, Raja Naeem Akram, and Damien Sauveron. Secure mobile edge server placement using multi-agent reinforcement learning. *Electronics*, 10(17):2098, 2021.
30. Yingya Guo and Cen Chen. DRLO: Optimizing edge server placement in dynamic MEC scenarios using deep reinforcement learning. *Computer Networks*, 268:111377, 2025.
31. Parviz Keshavarz Haddadha, Mohammad Hossein Rezvani, Mahdi MollaMotalebi, and Achyut Shankar. Machine learning methods for service placement: a systematic review. *Artificial Intelligence Review*, 57(3):61, 2024.
32. Jinxiang Sha, Jintao Wu, Mingliang Wang, Yonglin Pu, Sheng Lu, and Muhammad Bilal. Qoe-aware edge server placement in mobile edge computing using an enhanced genetic algorithm. *International Journal of Intelligent Networks*, 6:65–78, 2025.
33. Subrat Prasad Panda, Kaustabha Ray, and Ansuman Banerjee. Service allocation/placement in multi-access edge computing with workload fluctuations. In *Service-Oriented Computing – ICSOC 2021*, pages 747–755. Springer, 2021.
34. Penglin Dai, Biao Han, Ke Li, Xincao Xu, Huanlai Xing, and Kai Liu. Joint optimization of device placement and model partitioning for cooperative dnn inference in heterogeneous edge computing. *IEEE Transactions on Mobile Computing*, 24(1):210–226, 2025.
35. Guangming Cui, Qiang He, Xiaoyu Xia, Feifei Chen, Hai Jin, and Yun Yang. Robustness-oriented k edge server placement. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 81–90, 2020.
36. Kahlan Aljobory and Mehmet Akif Yazici. Edge server selection with round-robin-based task processing in multiserver mobile edge computing. *Sensors*, 25(11):3443, 2025.
37. Meena Rani, Kalpna Guleria, and Surya Narayan Panda. Enhancing performance and latency optimization in fog computing with a smart job scheduling approach. *Statistics, Optimization & Information Computing*, 13(1):309–330, 2025.
38. Qian Li, Yiguang Bai, and Weifeng Gao. Improved initialization method for metaheuristic algorithms: A novel search space view. *IEEE Access*, 9:121366–121384, 2021.
39. Amr H. Abdelhaliem, Islam S. Fathi, and Mohammed Tawfik. Fast and efficient feature selection in AI application based on enhanced binary secretary bird optimization algorithm. *Statistics, Optimization & Information Computing*, 14(5):2643–2662, 2025.
40. Ali Akbar Vali, Sadoon Azizi, and Mohammad Shojafar. Resp: A recursive clustering approach for edge server placement in mobile edge computing. *ACM Transactions on Internet Technology*, 24(3):1–25, 2024.
41. Sevda Zarei, Sadoon Azizi, and Awder Ahmed. Optimizing edge server placement and load distribution in mobile edge computing using aco and heuristic algorithms. *The Journal of Supercomputing*, 81(1):257, 2025.
42. Arman Sanaei Asl and Massoud Reza Hashemi. Resilient and efficient edge server placement in mobile edge computing: A dual-strategy approach for optimized performance and fault tolerance. *The Journal of Supercomputing*, 81(14):1319, 2025.
43. Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, New York, 1975.
44. János Sztrik. *Finite-Source Queueing Systems and their Applications*. University of Debrecen, Debrecen, Hungary, 2004.
45. John D. C. Little. A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383–387, 1961.
46. R.T. Marler and J.S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.
47. Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Kluwer Academic Publishers, Boston, MA, 1999.
48. Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia tools and applications*, 80(5):8091–8126, 2021.

49. Diogo Freitas, Luiz Guerreiro Lopes, and Fernando Morgado-Dias. Particle swarm optimisation: a historical review up to the current developments. *Entropy*, 22(3):362, 2020.