POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Breaking the Limits of Community Saturation in NEPTUNE: Novel Methods for Optimized Function Placement in MEC

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Alessandro Mosconi**

Student ID: 10681624
Advisor: Prof. Luciano Baresi
Academic Year: 2024-25

Abstract

Efficient resource allocation in Multi-Access Edge Computing (MEC) infrastructures poses a significant challenge, particularly when dealing with dynamic workloads and latency-sensitive applications. *NEPTUNE* [2], a framework developed at Politecnico di Milano, was designed to optimize the placement of serverless functions within MEC environments while addressing the constraints of limited resources.

This thesis focuses on the issue of community saturation in MEC infrastructures and proposes three resolution strategies: the first is based on a clustering algorithm with shared boundary nodes, allowing node overlap between adjacent communities to increase flexibility in resource usage; the second adopts a balanced clustering approach that considers computational capacity rather than just node count, promoting a more equitable workload distribution; the third introduces the EF-TTC (Edge FaaS-Top Trading Cycles) algorithm [14], which optimizes serverless function placement to enable larger community formations and reduce fragmentation.

Through experimental evaluation, the three solutions are analyzed in terms of response time, computational complexity, and proximity to the optimal result. The findings demonstrate a reduction in latency, more efficient use of resources, and better overall performance compared to conventional approaches.

Keywords: Multi-Access Edge Computing, serverless computing, clustering algorithms, resource management, function placement, NEPTUNE.

Abstract in lingua italiana

L'allocazione efficiente delle risorse nelle infrastrutture di Multi-Access Edge Computing (MEC) rappresenta una sfida cruciale, soprattutto nella gestione di carichi di lavoro dinamici e applicazioni sensibili alla latenza. *NEPTUNE* [2], un framework sviluppato presso il Politecnico di Milano, è stato progettato per ottimizzare il posizionamento delle funzioni serverless all'interno di ambienti MEC, affrontando i vincoli imposti dalla limitatezza delle risorse.

Questa tesi analizza in particolare il problema della saturazione delle comunità MEC e propone tre strategie risolutive: la prima si basa su un algoritmo di clustering con nodi condivisi ai confini, che consente la sovrapposizione di nodi tra comunità adiacenti per una maggiore flessibilità nell'uso delle risorse; la seconda adotta un clustering bilanciato fondato sulla capacità computazionale dei nodi, piuttosto che sul loro numero, promuovendo una distribuzione più equa dei carichi; la terza introduce l'algoritmo EF-TTC (Edge FaaS-Top Trading Cycles) [14], che ottimizza il posizionamento delle funzioni serverless, consentendo la formazione di comunità più grandi e una riduzione della frammentazione.

Attraverso un'analisi sperimentale, le tre soluzioni sono state valutate in termini di tempo di risposta, complessità computazionale e vicinanza al risultato ottimale. I risultati evidenziano una riduzione della latenza, un uso più efficiente delle risorse e migliori prestazioni rispetto agli approcci convenzionali.

Parole chiave: Multi-Access Edge Computing, serverless computing, algoritmi di clustering, gestione delle risorse, function placement, NEPTUNE.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
1.1 Context 1
1.2 Goals of the Thesis 2
1.3 Structure of the Thesis 2

2 The NEPTUNE Framework **5**
2.1 Overview 5
2.2 Neptune architecture 6
2.3 Design Principles and Components 7
2.4 Implementation Technologies 8
2.5 Limitations and Challenges 9
2.6 Motivation for Enhancements 11

3 State of the Art **13**
3.1 Clustering with Overlapping Boundary Nodes 13
3.2 Resource-Balanced Clustering 15
3.3 Inter-Community Load Balancing (Function Offloading) 16
3.4 Advanced Function Placement Algorithms 16
3.5 Predictive Models for Load Management 17
3.6 Comparative Analysis 18

4 Problem Definition and Proposed Solutions **21**
4.1 Problem Definition 21
4.1.1 Function Placement and Resource Constraints 21

4.1.2 Community Saturation Scenario . . . 21
4.2 Proposed Solutions . . . 23
4.2.1 Clustering with Overlapping Boundary Nodes . . . 23
4.2.2 Resource-Aware Balanced Clustering . . . 23
4.2.3 Function Placement Optimization and Computational Complexity . 24
4.3 Synthesis of Approaches . . . 25

5 Research Methodology 27
5.1 Naive Community Rerouting . . . 28
5.2 Heap-Based Clustering with Overlapping Nodes . . . 28
5.2.1 Main Idea . . . 28
5.2.2 SLPA Baseline Analysis . . . 29
5.2.3 Novel Algorithm . . . 29
5.3 Heap-Based Resource-Aware Clustering with Post-Balancing . . . 37
5.3.1 Main Idea . . . 37
5.3.2 SLPA Baseline Analysis . . . 38
5.3.3 Novel Algorithm . . . 38
5.4 EF-TTC Function Placement . . . 49
5.4.1 MIP Baseline Analysis . . . 49
5.4.2 Constraint Satisfaction and Variable Semantics . . . 50
5.4.3 Adapting EF-TTC to the MIP Framework . . . 51
5.4.4 Objective Functions and Local Scoring in EF-TTC . . . 51
5.4.5 Cycle Detection and TTC Application . . . 54
5.4.6 Convergence Strategy . . . 55
5.4.7 EF-TTC Algorithm: Design and Complexity Analysis . . . 55
5.4.8 Preliminary Observations . . . 62
5.5 Evaluation Metrics . . . 64

6 Results Analysis 65
6.1 Clustering Algorithms . . . 65
6.1.1 Evaluation Metrics . . . 65
6.1.2 Experimental Setup . . . 66
6.1.3 Cluster Compactness . . . 70
6.1.4 Computation Time . . . 72
6.1.5 Visual Inspection . . . 75
6.2 Function Placement with EF-TTC . . . 81
6.2.1 Evaluation Metrics . . . 81
6.2.2 Experimental Setup . . . 81

6.2.3 Computation Time 83
6.2.4 Objective Score 89
6.2.5 Constraint Satisfaction 91
6.3 Summary of Results 93

7 Conclusions and future developments **95**

Bibliography **99**

1 | Introduction

1.1. Context

The growing adoption of Multi-Access Edge Computing (MEC) frameworks has enabled the deployment of low-latency, high-performance applications closer to end-users. Unlike traditional cloud infrastructures, MEC systems operate under constraints such as limited resources, rapidly changing workloads, and geographic dispersion. NEPTUNE is a serverless-based framework [2] that addresses these challenges by optimizing function placement and dynamically allocating resources within independently managed clusters of edge nodes, known as communities.

To achieve this, NEPTUNE [2] employs a hierarchical control architecture with three coordinated layers. The **Topology Level** partitions the infrastructure into communities (groups of nearby nodes defined according to latency and connectivity metrics) using a clustering algorithm during the initial setup. These partitions remain static unless explicitly reconfigured. The **Community Level** oversees each group autonomously, using Mixed Integer Programming (MIP)[21] to determine the optimal placement of functions across the available nodes. At the lowest level, the **Node Level** performs fine-grained resource management on individual nodes through lightweight control-theoretic mechanisms for CPU and GPU allocation. Together, these layers allow NEPTUNE to adapt to resource contention and workload variability while ensuring the Quality of Service (QoS) demanded by edge applications.

One of the most critical challenges in MEC environments is **community saturation**. This condition arises when the computational resources of a community are no longer sufficient to efficiently handle incoming requests. The problem is amplified by NEPTUNE's reliance on fixed community structures and the high computational complexity of its placement algorithm [2]. As a result, latency increases, service quality deteriorates, and the system may fail to allocate new functions despite the availability of resources in other parts of the infrastructure.

This problem is especially significant in scenarios involving latency-sensitive applications

such as autonomous vehicles, AR/VR systems, and real-time analytics, highlighting the need for more flexible and scalable approaches to function placement and resource sharing within and across communities.

1.2. Goals of the Thesis

This thesis aims to identify, design, and evaluate scalable strategies to mitigate the impact of community saturation within the NEPTUNE framework [2]. The research activities are structured as follows:

- **Analysis of the State of the Art**: A comprehensive study of existing methodologies for clustering, function placement, and load balancing in MEC and serverless environments.
- **Review and Evaluation of NEPTUNE's Original Design**: Focused on assessing the strengths and limitations of the current community partitioning logic and function placement mechanism.
- **Design and Implementation of New Algorithms**: Three novel strategies are proposed and developed:
  1. Clustering with overlapping boundary nodes.
  2. Resource-aware balanced clustering.
  3. Efficient function placement algorithm based on the Edge FaaS-Top Trading Cycle (EF-TTC) model.
- **Experimental Evaluation**: The proposed algorithms are evaluated in simulated MEC environments using synthetic workloads. The analysis focuses on response time, algorithmic complexity, and the consistency of the results with respect to the baseline solution.

The outcomes of this research are intended to extend NEPTUNE's capabilities, improving its resilience to workload fluctuations and enabling more efficient and scalable deployment of edge applications.

1.3. Structure of the Thesis

This thesis is structured as follows:

- **Chapter 2: The NEPTUNE Framework** – Introduces the NEPTUNE frame-

work in detail, explaining its hierarchical control architecture, core components, and limitations in handling community saturation. This chapter ensures that the thesis is self-contained and understandable without requiring prior knowledge of NEPTUNE.

- **Chapter 3: State of the Art** – Reviews existing methodologies related to overlapping node clusters, resource-balanced clustering, advanced function placement algorithms, and predictive models for load management. It also includes strategies for inter-community load balancing.
- **Chapter 4: Problem Definition and Proposed Solutions** – Defines the specific problem of community saturation within MEC environments and presents the three proposed solutions: clustering with overlapping boundary nodes, resource-aware balanced clustering, and function placement optimization using the EF-TTC algorithm.
- **Chapter 5: Research Methodology** – Details the design and implementation of the proposed solutions, including naive rerouting, clustering algorithm modifications, and the adaptation of EF-TTC within NEPTUNE's architectural model.
- **Chapter 6: Results Analysis** – Presents the experimental evaluation of each proposed approach through simulation-based metrics such as response time, resource efficiency, and computational complexity, with comparison to the baseline NEPTUNE implementation.
- **Chapter 7: Conclusions and Future Developments** – Summarizes the research findings and discusses possible directions for extending the work, including implementation of the proposed strategies in the actual NEPTUNE codebase.

2 | The NEPTUNE Framework

2.1. Overview

NEPTUNE is a serverless-based framework designed to manage the deployment and execution of latency-sensitive applications in large-scale Multi-access Edge Computing (MEC) environments [2]. MEC extends cloud computing capabilities to the edge of the network, closer to end-users, by deploying computation and storage resources at or near base stations, access points, or other edge nodes. This proximity drastically reduces network delay and improves response times for time-sensitive services.

Unlike traditional cloud computing infrastructures, where computational resources are centralized and often physically distant from users, MEC nodes are inherently constrained in resources but benefit from their geographic closeness. This architecture is especially suited for scenarios where real-time performance is critical. For instance, in autonomous driving, the decision-making process must occur in milliseconds to ensure safety. Similarly, AR/VR applications require low latency to avoid user disorientation, and real-time analytics benefit from faster local processing.

An illustrative representation of a typical MEC deployment scenario is provided in Figure 2.1, where edge nodes handle requests closer to users compared to a centralized cloud.

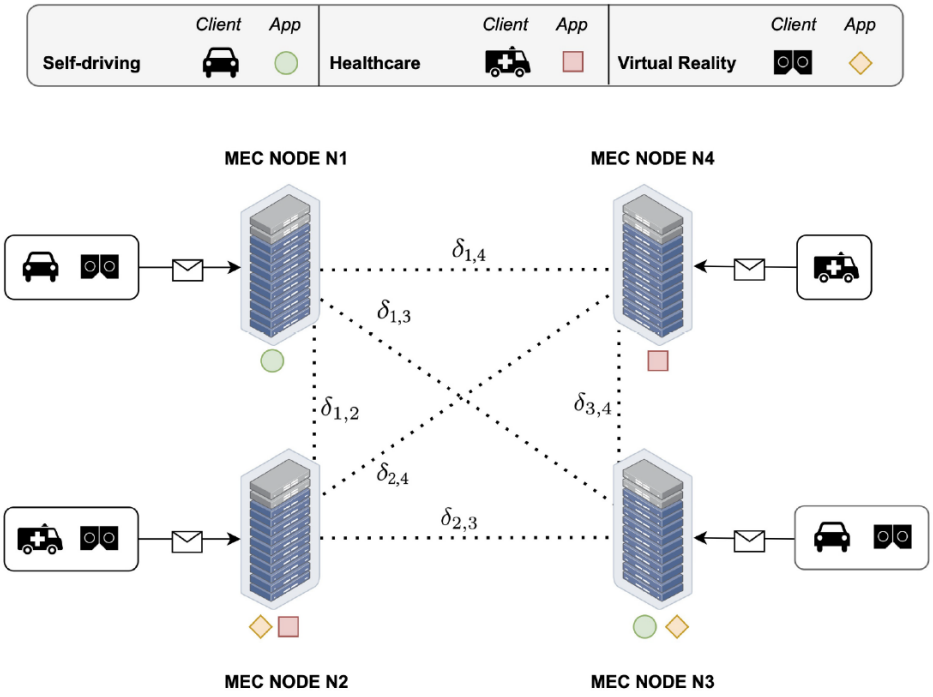

Figure 2.1: Example of a Multi-Access Edge Computing (MEC) deployment with edge nodes positioned near end-users. Image adapted from "*NEPTUNE - A Comprehensive Framework for Managing*" [2].

2.2. Neptune architecture

The core objective of NEPTUNE is to enable the efficient placement and management of serverless functions in MEC systems while respecting latency constraints and avoiding resource contention. To tackle this, NEPTUNE adopts a hierarchical control architecture composed of three interconnected levels, each with a distinct role in managing infrastructure complexity and ensuring scalability:

- **Topology Level**: This top-level component partitions the entire MEC infrastructure into latency-aware clusters called "communities." It employs the Speaker-Listener Label Propagation Algorithm (SLPA), which forms communities based on measured inter-node network delays and geographic proximity. The goal is to ensure that all nodes within a community can communicate within a bounded delay, allowing for local processing decisions.
- **Community Level**: Each community is independently managed by a dedicated controller that handles function placement and routing policies. Using a Mixed Integer Programming (MIP) formulation, this level seeks to optimize two competing

objectives: (i) minimizing network latency by placing function instances near the request origin, and (ii) reducing the number of active nodes to conserve resources. The algorithm accounts for heterogeneous resources, such as CPU and GPU availability.

- **Node Level**: At the granularity of individual MEC nodes, NEPTUNE applies control-theoretic techniques to adjust resource allocations dynamically. Specifically, Proportional-Integral (PI) controllers monitor system load and scale resources vertically, while a contention manager ensures fair CPU scheduling among multiple function instances to avoid overload.

This architecture is illustrated in Figure 2.2, where each control level and its components are shown in context, from global topology management down to node-level control loops.

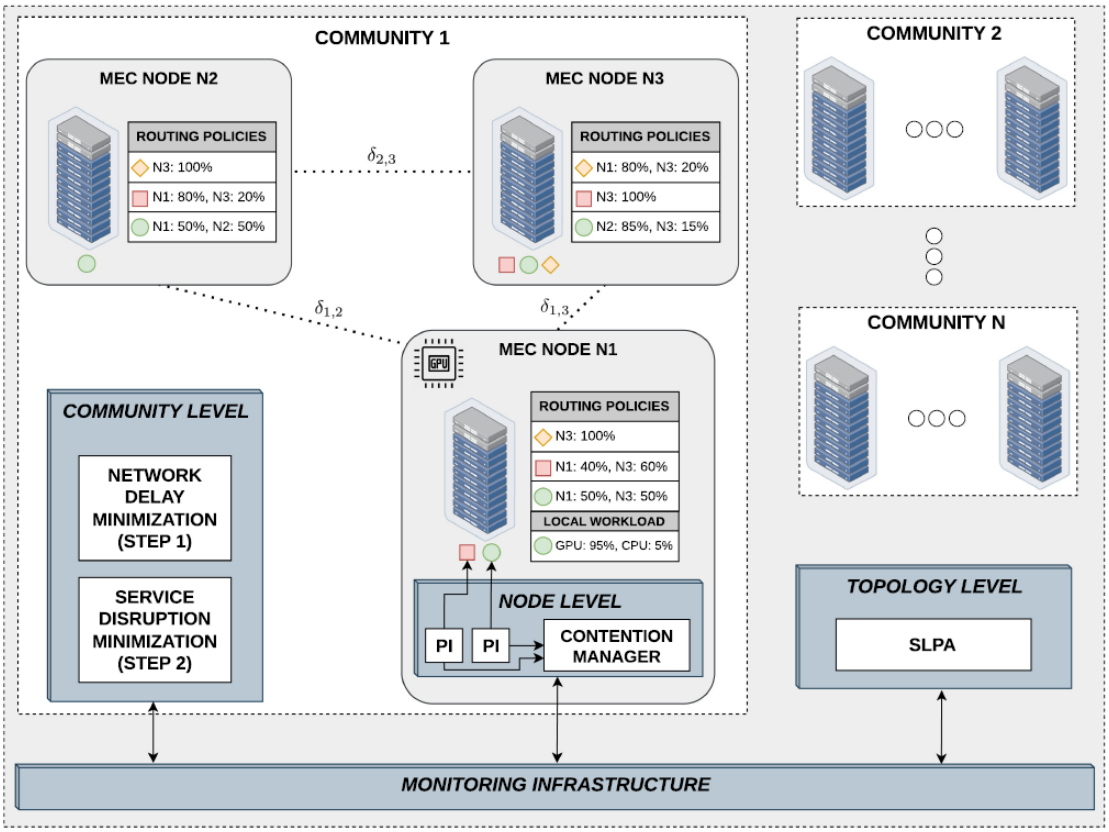

Figure 2.2: NEPTUNE hierarchical architecture. Image adapted from "*NEPTUNE - A Comprehensive Framework for Managing*" [2] .

2.3. Design Principles and Components

NEPTUNE assumes that applications are decomposed into serverless functions, each encapsulated in a lightweight container. Users define these functions along with their re-

quired memory and response time constraints. NEPTUNE then automatically deploys and manages the functions according to workload conditions and resource availability.

Key features of the system include:

- **Latency-Aware Placement**: Functions are placed on nodes that are geographically close to users, reducing round-trip delays.
- **Dynamic Resource Allocation**: Resources are allocated dynamically, enabling the system to react to workload spikes in real-time.
- **Cold Start and Graceful Termination Handling**: The placement algorithm accounts for cold start delays and ensures that ongoing requests are not interrupted during function migration.

2.4. Implementation Technologies

The NEPTUNE framework has been implemented by integrating and extending widely adopted open-source technologies, with the goal of providing an efficient, modular, and scalable orchestration platform for edge-based serverless computing.

At its core, NEPTUNE is built on **K3s**, a lightweight Kubernetes distribution optimized for edge environments. K3s manages container orchestration across geo-distributed and resource-constrained MEC nodes. Serverless applications are deployed using **OpenFaaS**, which allows functions to be defined as lightweight containers and managed through standard Kubernetes interfaces.

NEPTUNE introduces several *Custom Resource Definitions (CRDs)*, including `Function`, `ServiceLevelAgreement`, and `CommunitySchedule`, to express deployment requirements, response time constraints, and routing policies. Each CRD is handled by a dedicated controller associated with one of the three architectural levels: `TopologyController`, `CommunityController`, and `NodeController`. These controllers interpret the CRDs to perform clustering, placement optimization, and resource scaling accordingly.

Fine-grained monitoring is achieved using the **sidecar pattern**: each function pod contains an auxiliary proxy container that intercepts incoming requests and pushes latency and workload metrics to a centralized *Metrics Database*. This approach enables tight feedback control while maintaining separation of concerns.

The prototype uses **Kubernetes Enhancement Proposal (KEP) 1287** to enable *in-place vertical scaling*, avoiding function restarts during CPU resource updates. This is crucial to promptly adapt to bursty workloads without service disruption.

To support GPU sharing, NEPTUNE relies on a custom version of the **AWS Virtual GPU Device Plugin**, which exposes physical GPUs as multiple virtual devices. GPU allocation is managed through **Nvidia Multi-Process Service (MPS)**, allowing containers to concurrently access the same physical accelerator.

Additionally, a **custom load balancer** is deployed on each node to enforce routing policies. Unlike standard solutions, it supports seamless updates without restarting services and reflects new placements computed by the Community controller.

All components are integrated into a fully working prototype. The source code and workload generator are publicly available at: `https://github.com/deib-polimi/neptune`

2.5. Limitations and Challenges

Despite its effectiveness, the NEPTUNE framework presents several limitations that can hinder its scalability, adaptability, and overall performance in dynamic and large-scale Multi-access Edge Computing (MEC) environments:

- **Fixed Community Structure**: The initial clustering of edge nodes into communities is performed statically using algorithms such as the Speaker-Listener Label Propagation Algorithm (SLPA), based on topological and latency criteria. However, once defined, these communities are not reconfigured at runtime. This rigid partitioning limits the system's flexibility in adapting to changing workload distributions, mobility patterns, or variations in resource availability. In highly dynamic environments, this static approach can lead to resource fragmentation and inefficient utilization, especially when some communities become overloaded while others remain underutilized.
- **Resource Saturation without Overflow Handling**: One of the most critical limitations in NEPTUNE is the absence of any native mechanism to handle cases where a community becomes saturated. When the resources (CPU, GPU, memory) within a community are fully utilized, NEPTUNE simply cannot place additional function instances in that region, and no fallback or overflow strategy is defined. This means that function placement requests may be dropped or significantly delayed, even if adjacent communities possess idle capacity. This constraint becomes particularly problematic in latency-sensitive applications, where service degradation due to function rejection or queuing can severely impact user experience and violate Quality of Service (QoS) requirements.
- **High Computational Complexity of the Placement Algorithm**: NEPTUNE

employs a Mixed Integer Programming (MIP) formulation to determine the optimal placement of functions within each community. While this approach guarantees high-quality placement decisions by jointly optimizing for latency, resource availability, and energy efficiency, it is computationally expensive. The solving time of the MIP model grows rapidly with the size of the problem, making it unsuitable for large infrastructures or scenarios with high-frequency reconfigurations. As a consequence, NEPTUNE's placement module may struggle to keep up with real-time demands, limiting the framework's applicability in edge environments characterized by fast and unpredictable workload fluctuations.

To illustrate the saturation problem discussed above, Figure 4.1 presents a conceptual visualization of a MEC infrastructure affected by community saturation. It shows how two communities, A and B, are represented: a new request is directed to Community A, which is already saturated. As a result, the request cannot be handled, even though Community B still has available resources. This highlights the inefficiency caused by the absence of inter-community coordination, which prevents load redistribution and leads to increased request rejection.

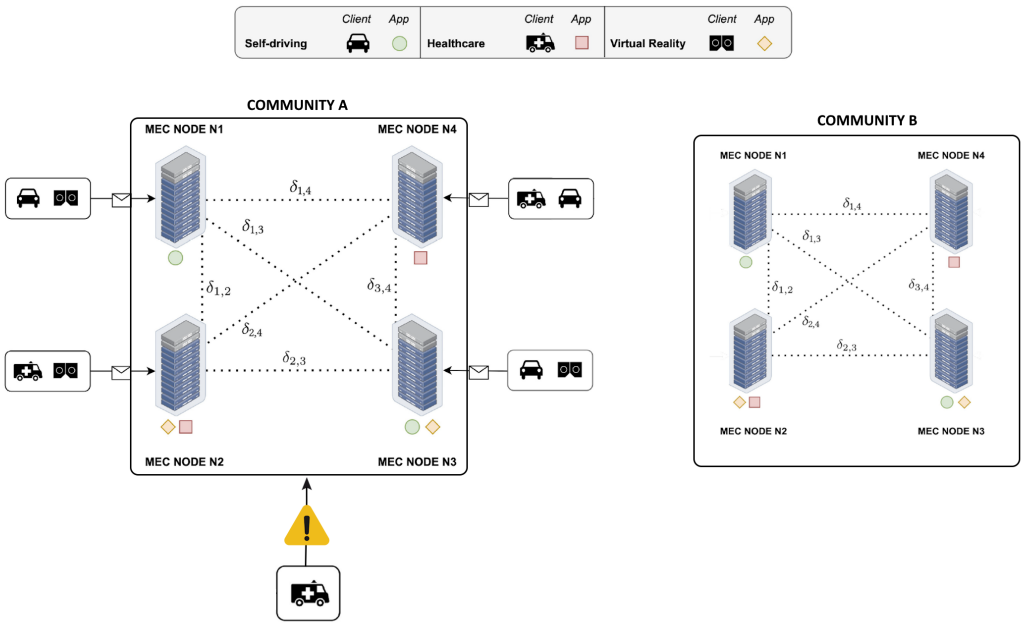

Figure 2.3: Example of resource saturation in MEC communities. Community A is saturated while Community B has available capacity. Due to the absence of inter-community overflow handling, new requests in A are rejected.

These challenges highlight the need for enhanced architectural features that allow dynamic reconfiguration, inter-community coordination, and computationally efficient placement

heuristics to better support MEC workloads in real-world deployments.

2.6. Motivation for Enhancements

These limitations motivate the need for enhanced strategies that improve the scalability and responsiveness of NEPTUNE. Specifically, mechanisms are needed to:

- Enable more flexible community structures that adapt to changing workloads.
- Share nodes or resources across communities to avoid saturation.
- Employ placement algorithms that strike a better balance between optimality and computational efficiency.

The remainder of this thesis explores these directions by proposing novel clustering and placement strategies aimed at overcoming NEPTUNE's current limitations.

3 | State of the Art

This chapter analyzes existing solutions for managing community saturation in distributed systems, with a focus on their applicability to Neptune. The review covers five main approaches: overlapping primary/reserve nodes, resource-balanced clustering, inter-community load balancing (function offloading), function placement algorithms, and predictive models for load management.

3.1. Clustering with Overlapping Boundary Nodes

This approach designates primary nodes to handle standard community workloads, while reserve nodes are shared among multiple communities and activated only during saturation. This structure increases flexibility, allowing temporary load distribution to neighboring nodes during peak times.

The *Speaker-Listener Label Propagation Algorithm (SLPA)*[18, 22] is a key solution for detecting overlapping communities:

- **Strengths:**
  - **Overlap Detection:** SLPA accurately identifies nodes belonging to multiple communities, enabling natural community overlap—ideal for systems where nodes share roles across groups.
  - **Computational Efficiency:** The time complexity of the entire algorithm is O(Tn) in sparse networks[22], for this reason SLPA scales well for large systems.
  - **Parameter Flexibility:** Customizable thresholds for overlap and iterations allow optimization for specific scenarios.
- **Limitations:**
  - **Threshold Dependency:** Performance varies significantly with overlap thresholds, requiring careful tuning.
  - **Non-Deterministic Outputs:** Stochastic interactions[22] may lead to incon-

sistent results across executions.

- **Fragmentation in Dense Networks:** Densely connected communities may become excessively fragmented, and there is no native mechanism to control cluster size.

SLPA is mature and validated for social networks, but its lack of native resource balancing and mechanism to control cluster size motivates the exploration of alternatives like *Balanced Label Propagation (BLP)*[20], discussed in Section 3.2, or a totally new custom algorithm.

In the context of node sharing and boundary management between communities, additional techniques can be applied to identify suitable nodes for overlapping assignment:

- **Distributed Boundary Detection (DBD)** [4]: this algorithm relies solely on local topology (up to three hops) to determine whether a node lies on an internal or external boundary. It is:
  - *Fully distributed*—each node takes decisions independently,
  - *Scalable*—suitable for large, dynamic edge networks,
  - *Mobility-tolerant*—maintains accuracy in mobile scenarios.

  However, it was originally designed for WSNs and may need adaptation for MEC environments. Additionally, it doesn't distinguish functional boundaries (e.g., inter-cluster vs. edge of network), requiring refinements for logical community overlaps.
- **Network Boundary Recognition via Graph-Theory** [7]: a centralized method using spectral tessellation and betweenness centrality to identify boundary nodes.
  - *Accurate* in distinguishing internal nodes and true bridges between communities,
  - *Based on global structural metrics*, enabling better identification of bridge/shared nodes.

  Its main drawbacks are high computational cost and reliance on centralized knowledge of the topology, which limit its usability in highly dynamic or large-scale edge systems.

Both approaches can inform the assignment of shared nodes between communities, either by enabling distributed identification (DBD) or by precise but centralized analysis (Graph-Theory). They could be integrated after a clustering algorithm to identify possible shared

nodes

3.2. Resource-Balanced Clustering

This approach clusters nodes while balancing computational resources (such as CPU and memory) to optimize workload distribution in MEC environments. Unlike traditional clustering algorithms that focus only on spatial proximity or number of nodes, resource-balanced clustering ensures that each cluster is formed with consideration of its total resource capacity, improving system efficiency and resilience to workload spikes.

Several methods from the literature can be adapted to support resource-based clustering:

- **Balanced K-Means** [13] is an extension of the classical k-means algorithm that incorporates cluster size constraints through the Hungarian assignment algorithm. Although originally intended to balance the number of nodes, it can be modified to constrain the total resource weight per cluster, such as CPU or memory availability.
- **Weighted Graph Clustering** [6, 12] constructs communities using node and edge weights that represent features like centrality or communication cost. When node weights are defined in terms of computational capacity, this approach allows the clustering process to prioritize balanced resource aggregation rather than uniform node distribution.
- **Significance-Based Detection** [17] employs hypothesis testing to detect statistically significant communities in weighted networks. Although computationally intensive, it can be applied to identify subgraphs where the internal resource availability deviates from a reference model, supporting the design of clusters that avoid saturation or under-utilization.
- **Fuzzy Clustering with Capacity Constraints** [3] allows each node to belong to multiple clusters, with soft membership values, and introduces explicit capacity constraints for each cluster. This formulation is particularly well-suited for environments with variable workloads, as it allows flexible resource assignments while guaranteeing that each cluster respects a global resource budget.
- **Balanced Label Propagation (BLP)** [20] is a scalable algorithm for partitioning large graphs while enforcing hard constraints on partition sizes. Although designed for balanced node count, the constraints can be reinterpreted as total resource thresholds. Additionally, its optimization of edge locality minimizes inter-cluster traffic, which is beneficial for latency-sensitive MEC scenarios.

These techniques provide a foundation for implementing clustering strategies that consider heterogeneous and dynamic resource availabilit.

3.3. Inter-Community Load Balancing (Function Offloading)

Function offloading is a key technique in managing saturated communities in serverless Multi-Access Edge Computing (MEC) systems. It enables the dynamic redistribution of functions to less loaded communities or nodes, alleviating resource bottlenecks. According to Ghorbian and Ghobaei-Arani [9], offloading strategies can be broadly classified into ML-based, heuristic-based, framework-based, and in-network computing (INC)-based approaches. Below are three representative solutions evaluated in recent literature:

- **MOSEC (Multi-Objective Deep RL)** [23]: A multi-objective reinforcement learning approach that jointly optimizes application completion time, energy consumption on user devices, and execution cost. It dynamically adapts to shifting user preferences and mitigates policy forgetting through Near-on Experience Replay and an Earliest-First scheduling mechanism. Suitable for complex and latency-sensitive serverless applications.
- **In-Network FaaS** [5]: Introduces the execution of serverless functions directly at edge nodes, reducing reliance on centralized cloud services. It explores the feasibility of stateful FaaS in decentralized environments, highlighting challenges related to state consistency, vendor lock-in, and orchestration across heterogeneous nodes.
- **MADDPG for Task Offloading** [11]: A multi-agent reinforcement learning algorithm for vehicular edge networks. Each vehicle acts as an agent and learns optimal offloading policies based on dynamic network states. The method significantly reduces latency and energy consumption, although scalability in dense environments remains an open issue.

3.4. Advanced Function Placement Algorithms

Optimal function placement [10] is critical for ensuring efficient use of computational resources and meeting QoS constraints in serverless edge computing. Traditional placement approaches struggle with dynamic workloads, cold start delays, and heterogeneous infrastructure. This section presents two notable methods that address these challenges with innovative mechanisms.

- **EF-TTC (Edge FaaS–Top Trading Cycles)** [14]: This algorithm models the function placement and scheduling problem using the School Choice paradigm and solves it through the Top Trading Cycles (TTC) mechanism. It aims to reduce execution time by favoring warm starts and aligning user preferences with edge server capabilities. EF-TTC is particularly effective in environments where minimizing cold start delays is crucial. Simulations show it outperforms traditional greedy and LFU-based approaches, though it requires accurate local resource state awareness to function properly.
- **EOPA/DCNOPA** [24]: These algorithms address virtual machine (VM) placement in MEC. EOPA performs exhaustive enumeration to find the placement that minimizes average data traffic, achieving optimal results at the cost of high computational complexity. DCNOPA offers a practical alternative by partitioning the problem into subproblems (one per VM), significantly reducing complexity while achieving near-optimal performance. These strategies are particularly relevant for function replication scenarios in large-scale networks.

Both approaches provide solid foundations for enhancing NEPTUNE's function placement module [2]. In particular, EF-TTC and DCNOPA offer significantly better computational complexity compared to NEPTUNE's current exponential-time placement logic. EF-TTC can improve latency and responsiveness in high-demand contexts, while DCNOPA enables scalable deployments by efficiently managing data traffic across a wide edge network

3.5. Predictive Models for Load Management

Predictive models for workload forecasting can enhance the reactivity and efficiency of MEC frameworks like NEPTUNE, enabling the anticipation of saturation conditions and more proactive resource allocation strategies.

- **Workload Prediction for Efficient Node Management in MEC** [16]: This approach integrates a heuristic selection algorithm with machine learning, specifically Support Vector Regression (SVR), to predict computational load at edge nodes. The model improves latency and load balancing, particularly in 5G/6G scenarios, by reducing unnecessary activations of service nodes and enabling proactive deployment of resources.
- **Multivariate LSTM-Based Location-Aware Prediction** [15]: By leveraging temporal and spatial correlations among edge data centers, this method uses a multivariate Long Short-Term Memory (LSTM) model to predict workloads. It

significantly improves accuracy (up to 44%) over non-location-aware baselines and supports timely scaling and service placement, albeit at the cost of higher computational complexity.

- **LSTM-Driven Optimal Workload Allocation** [19]: This technique forecasts application-specific load using LSTM and separates workload allocation into two sub-problems: task assignment and resource allocation. Simulation results demonstrate substantial reductions in response time (up to 60%) and average completion time, offering near-optimal performance with linear-time solvability in practice.

Despite their effectiveness, these predictive approaches demand constant data collection and training, making them more suited for long-term evolution of frameworks like NEPTUNE rather than immediate integration.

3.6. Comparative Analysis

Table 3.1 compares the main categories of approaches considered for addressing community saturation in NEPTUNE[2].

Approach	Strengths	Limitations	Suitability for NEPTUNE
Overlapping Clustering	Enables node sharing at boundaries and supports flexible redirection of load between communities	No built-in resource balancing; cluster size control must be implemented externally	Medium (requires algorithmic replacement or enhancement)
Resource-Aware Clustering	Ensures equitable distribution of computational resources across communities	Requires constraints on node count to limit placement complexity	Medium (requires algorithmic replacement)
Inter-Community Offloading	Dynamically redistributes load between communities; mitigates local overload	Adds orchestration overhead and requires runtime coordination logic	High (modular and complementary integration possible)
Advanced Placement Algorithms	Reduces placement latency and improves scalability with lower complexity than current MIP logic	Needs accurate local state and adaptation to NEPTUNE's scheduling logic	High (suitable as a replacement for current core logic)
Predictive Load Management	Enables proactive scheduling by anticipating saturation	High computational cost and need for continuous telemetry and retraining	Low (long-term enhancement potential)

Table 3.1: Comparison of proposed solutions for mitigating community saturation in NEPTUNE.

4 | Problem Definition and Proposed Solutions

4.1. Problem Definition

4.1.1. Function Placement and Resource Constraints

NEPTUNE[2] employs a Mixed Integer Programming (MIP) [21] strategy to determine optimal function placement in a distributed edge infrastructure. Given the NP-hard nature of the problem, the network is partitioned into communities of nodes. Each community is managed independently by a controller that assigns functions to nodes within its scope. This architecture ensures manageable computation times, but at the cost of limited global visibility and flexibility.

4.1.2. Community Saturation Scenario

Community saturation occurs when the available computational resources within a community are fully utilized, preventing the handling of additional function requests. This scenario typically results from workload imbalances, localized spikes in user activity, or limited resource-sharing mechanisms between communities. Within NEPTUNE's hierarchical architecture [2], each community autonomously manages function placement and scaling. While this design promotes scalability, it limits real-time inter-community coordination, increasing the likelihood of localized overloads, as shown in Figure 4.1.

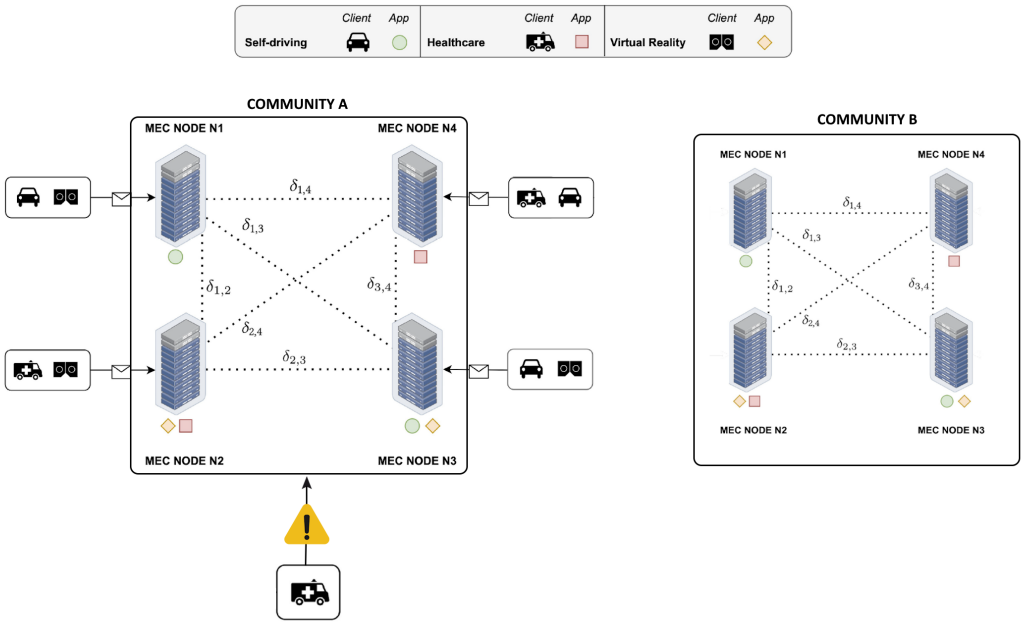

Figure 4.1: Example of resource saturation in MEC communities. Community A is saturated while Community B has available capacity. Due to the absence of inter-community overflow handling, new requests in A are rejected.

Saturation can lead to the following critical issues:

- **Increased response times due to queuing and rerouting**: Once a node reaches capacity, new requests are queued or redirected within the community. As saturation escalates, queues lengthen and response times exceed the SLA (Service Level Agreement).
- **Underutilization of neighboring community resources**: Due to NEPTUNE[2]'s localized control model, nearby communities may remain underloaded while a saturated community struggles, leading to inefficient global resource usage.
- **Service degradation or function failure**: When neither local nodes nor adjacent communities can process requests within acceptable time bounds, services may degrade or fail entirely. This is especially problematic for latency-sensitive applications such as real-time control, augmented reality, or autonomous systems.

In NEPTUNE's current implementation [2], function requests that arrive at a saturated community are simply dropped. A basic rerouting mechanism could alleviate this issue, but the limitations of fixed clustering and decentralized placement strategies may still result in service degradation.

Ultimately, the absence of global coordination, coupled with rigid community partitioning

and complex placement logic, makes NEPTUNE [2] vulnerable to saturation scenarios that compromise both performance and reliability.

4.2. Proposed Solutions

4.2.1. Clustering with Overlapping Boundary Nodes

This approach extends conventional clustering by allowing specific nodes located at the boundaries between communities to belong to multiple clusters. In the context of NEPTUNE [2], where communities are used to localize computation, the use of rigid and disjoint partitions may limit the system's adaptability. Introducing overlap enables each community to manage two distinct sets of nodes: a set of exclusive nodes, used as the primary resource pool, and a set of shared nodes, which are accessible by multiple communities. In the presence of variable workloads or saturation, a community can fall back on available shared nodes, thereby increasing placement flexibility and improving overall resource utilization across the edge infrastructure.

Design Objectives

- Allow the sharing of boundary nodes between communities without violating latency or QoS constraints.
- Ensure that each cluster maintains a sufficient number of exclusive nodes to guarantee operational autonomy.
- Guarantee the presence of a minimum number of shared nodes per cluster to support controlled and predictable resource sharing in case of saturation.

Implementation Choice

A custom algorithm was developed, inspired by k-means clustering [1]. After the initial clustering phase, the nodes furthest from the centroid of each cluster are identified and marked as boundary nodes. These nodes are then designated as shared across adjacent communities to increase flexibility and promote load redistribution. The algorithm is described in detail in Section 5.2

4.2.2. Resource-Aware Balanced Clustering

This strategy redefines the clustering process by prioritizing available computational capacity rather than the number of nodes. The aim is to create communities that are

balanced in terms of processing resources, rather than purely in size.

Design Objectives

- Incorporate hardware heterogeneity into the clustering logic.
- Form communities with balanced computational capacity.
- Prevent saturation by promoting an even distribution of workloads.

Implementation Choice

A modified k-means-based algorithm [1] was implemented, introducing constraints that limit the total amount of resources assigned to each cluster, ensuring that this total tends toward the average availability across the entire infrastructure (i.e., total resources divided by the number of clusters). Following the initial assignment phase, a rebalancing step is performed, during which nodes close to cluster boundaries may be reassigned to neighboring clusters. This adjustment aims to preserve cluster compactness while aligning each cluster's total resource sum as closely as possible to the global average. A detailed description of the algorithm is provided in Section 5.3

4.2.3. Function Placement Optimization and Computational Complexity

The current function placement strategy in NEPTUNE [2] relies on a Mixed Integer Programming (MIP) [21] formulation, which is known to be NP-complete. While this guarantees high-quality placement decisions, it also introduces significant computational overhead, making the approach unsuitable for real-time or large-scale deployments. Replacing the MIP solver with an algorithm of polynomial complexity would greatly reduce execution time and enable faster adaptation to dynamic workloads.

Improving placement efficiency in this way would allow the formation of larger edge communities without compromising responsiveness, thereby reducing the likelihood of saturation events. In an ideal scenario, a sufficiently fast placement algorithm could eliminate the need for strict community partitioning altogether by enabling centralized placement over the entire edge infrastructure. Although such a level of scalability is not yet achievable, reducing complexity remains a critical step toward that direction.

Design Objectives

- Reduce the computational overhead associated with function placement to enable real-time operation.
- Improve scalability by supporting larger communities with unified placement logic.
- Minimize saturation events by enabling more flexible and efficient resource utilization across the edge.

Implementation Choice

A modified version of the Edge FaaS–Top Trading Cycles (EF-TTC) [14] algorithm was implemented to better align with NEPTUNE's requirements [2]. The adaptation focuses on preserving responsiveness and scalability while integrating system-specific constraints related to function cold starts, node availability, and policy enforcement within communities. Section 5.4 outlines the algorithm in detail.

4.3. Synthesis of Approaches

The solutions proposed in this work address complementary aspects of the scalability and saturation issues in NEPTUNE-based[2] MEC infrastructures:

- **Clustering with overlapping boundary nodes** enhances the flexibility of community boundaries, allowing selected nodes to be shared across communities and improving adaptability under variable workloads.
- **Resource-aware balanced clustering** refines the initial partitioning of the infrastructure by considering computational capacity, rather than node count alone, as the main criterion for cluster formation.
- **Complexity-aware placement strategies** replace the current MIP-based [21] placement method with lower-complexity algorithms, enabling real-time decisions and supporting larger communities that can react more effectively to dynamic workloads.

These three approaches are modular and can be composed to reinforce one another: for example, resource-aware clustering can generate communities where fast placement algorithms operate more efficiently, while overlapping nodes can serve as shared fallback resources to mitigate temporary overloads without requiring full community reconfiguration.

5 | Research Methodology

This chapter presents the methodological approach adopted to investigate potential solutions to the problem of resource saturation in edge computing communities within the NEPTUNE framework [2]. The research explores four strategies, ordered by increasing complexity and impact on the overall system: (i) trivial rerouting of requests to nearby communities, (ii) overlapping-node clustering with structural constraints, (iii) resource-aware balanced clustering, and (iv) enhancement of NEPTUNE's placement logic via an improved function allocation strategy known as EF-TTC (Edge Function Topology-Traffic Cycles)[14].

The clustering algorithms are based on the construction of efficient, balanced clusters by leveraging proximity information and priority queues. While the method does not follow a classical agglomerative approach, it conceptually resembles the strategy introduced in *"Fast Agglomerative Clustering Using a k-Nearest Neighbor Graph"* [8], where local neighborhood information and efficient data structures are utilized to reduce the computational complexity of hierarchical clustering. Although the original method is based on constructing a k-nearest neighbor (k-NN) graph to limit the search space, the inspiration taken in this work lies specifically in the use of efficient heap-based mechanisms. In the proposed algorithm, a min-heap is maintained for each cluster centroid, containing all unassigned nodes sorted by increasing distance from the centroid. This allows for efficient and scalable selection of the closest nodes to be added to each cluster without needing to precompute or store a k-NN graph. The strategy effectively reduces the complexity of the clustering process while preserving locality, and contributes to the formation of compact and well-balanced clusters. Furthermore, the choice of explicitly selecting and updating cluster centroids during the clustering process is inspired by the k-means algorithm [1], where centroids act as attractors to group data points based on spatial proximity, ensuring coherence and structure in the resulting clusters.

5.1. Naive Community Rerouting

As a baseline, I initially considered a straightforward request rerouting mechanism. In this model, when a given community becomes saturated, incoming requests are simply redirected to the nearest available community. While effective in bypassing overload conditions, this solution offers no guarantees on performance optimization or latency minimization and fails to provide long-term load balancing. Due to these limitations, this strategy is only considered as a fallback mechanism and was not analyzed in depth.

5.2. Heap-Based Clustering with Overlapping Nodes

The first substantive solution investigated was the use of clustering algorithms that support community overlaps. Initially, I explored the SLPA (Speaker-listener Label Propagation Algorithm)[18, 22], which inherently supports overlapping clusters. However, SLPA's probabilistic nature and lack of support for structural constraints, such as a minimum number of exclusive/shared nodes per cluster, made it inadequate for our needs.

To overcome these limitations, I designed a custom clustering algorithm grounded in deterministic logic and optimized heap-based data structures. This algorithm allows for:

- Balanced population of clusters based on centroid proximity.
- Enforcement of user-defined constraints on exclusive and shared nodes.
- Reproducible, deterministic clustering through metric-driven decisions rather than probabilistic inference.

5.2.1. Main Idea

The use of overlapping nodes between clusters provides additional flexibility in balancing the load across communities. When a cluster is close to saturation, shared nodes act as a buffer, allowing for partial load redirection without violating exclusivity constraints. Furthermore, the hard enforcement of a minimum number of exclusive and shared nodes ensures that each community has a stable core while maintaining collaborative links to adjacent communities. This structural arrangement supports more robust function placement and request redirection in resource-constrained environments.

5.2.2. SLPA Baseline Analysis

The Speaker-listener Label Propagation Algorithm (SLPA)[18, 22] is a well-known method for overlapping community detection that relies on probabilistic interactions to propagate labels. Despite its ability to produce overlapping clusters, SLPA suffers from non-deterministic behavior and lacks support for structural constraints. In particular, it does not allow for enforcing a minimum number of shared or exclusive nodes per community.

In the Neptune framework [2], SLPA was modified to discard the overlapping capability and introduce hard constraints on community size. This adaptation ensures more predictable outcomes but fundamentally removes the flexibility enabled by shared nodes. By contrast, the overlapping clustering algorithm we propose introduces a controlled sharing mechanism: it explicitly enforces constraints on both shared and exclusive nodes per cluster, enabling flexible boundary management while maintaining structural guarantees.

5.2.3. Novel Algorithm

The procedure consists of three major stages: (i) centroid initialization (Algorithm 5.2), (ii) balanced population via heaps (Algorithm 5.3), and (iii) post-processing to enforce the minimum number of shared nodes per cluster (Algorithm 5.4). The overall process is summarized in Algorithm 5.1.

Algorithm 5.1 Main Overlapping Clustering Algorithm

Require: Distance matrix D, $maxNodesPerCluster$, $minSharedNodes$, $minExclusiveNodes$

Ensure: Set of balanced clusters

$n \leftarrow$ number of nodes
$k \leftarrow \lceil n/maxNodesPerCluster \rceil$
$clusterSize \leftarrow maxNodesPerCluster$
if $minSharedNodes + minExclusiveNodes \geq clusterSize$ **then**
    error "Impossible constraints"
$centroids \leftarrow$ InitializeCentroids(D, k)
$clusters \leftarrow$ array of k empty sets
for $i \leftarrow 1$ to k **do**
    $clusters[i] \leftarrow clusters[i] \cup \{centroids[i]\}$
PopulateClusters(D, $clusters$, $centroids$, $clusterSize$)
EnforceSharedNodes(D, $clusters$, $centroids$, $minSharedNodes$, $minExclusiveNodes$)
return $clusters$

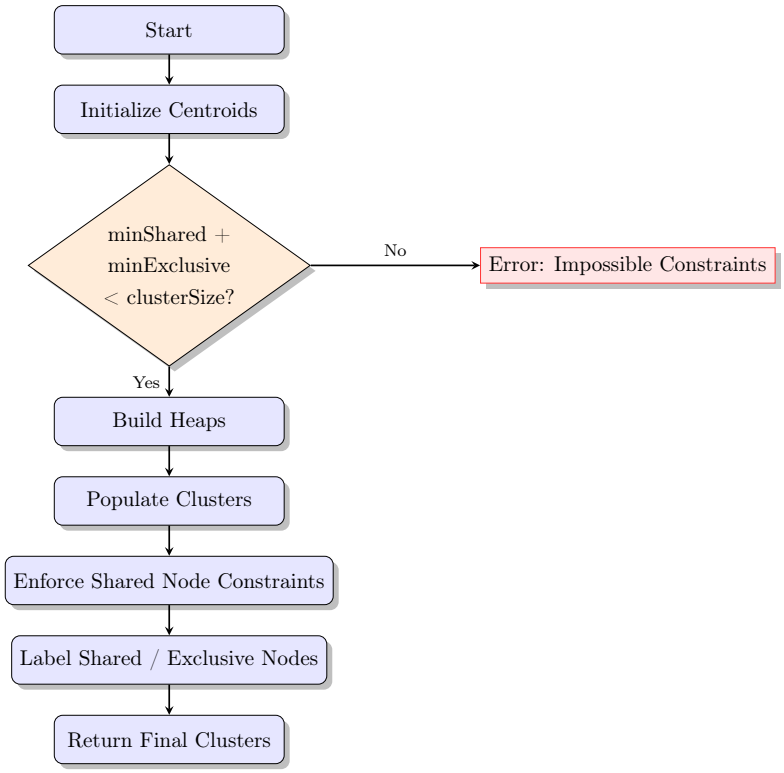

Figure 5.1: Flowchart of the overlapping clustering algorithm.

1. Centroid Initialization The centroid selection phase (Algorithm 5.2) employs a greedy strategy to ensure maximal separation between initial cluster seeds. The first centroid is chosen arbitrarily. Subsequent centroids are selected iteratively using a greedy strategy: for each step, the node farthest from all previously chosen centroids (in terms of minimal pairwise distance) is selected. This is implemented efficiently by caching the minimum distances and updating them iteratively.

This stage has a complexity of $O(k \cdot n)$, where k is the number of clusters and n the number of nodes.

Algorithm 5.2 Centroid Initialization

Require: Distance matrix D, number of clusters k

Ensure: Set of k centroids

$centroids \leftarrow \{node_1\}$ ▷ Select first node

$distances \leftarrow$ array of n values ∞

while $|centroids| < k$ **do**

  $last \leftarrow$ last element of $centroids$

  for each node v not in $centroids$ **do**

    $distances[v] \leftarrow \min(distances[v], D[last][v])$

  $newCentroid \leftarrow$ node with maximum $distances[v]$

  $centroids \leftarrow centroids \cup \{newCentroid\}$

return $centroids$

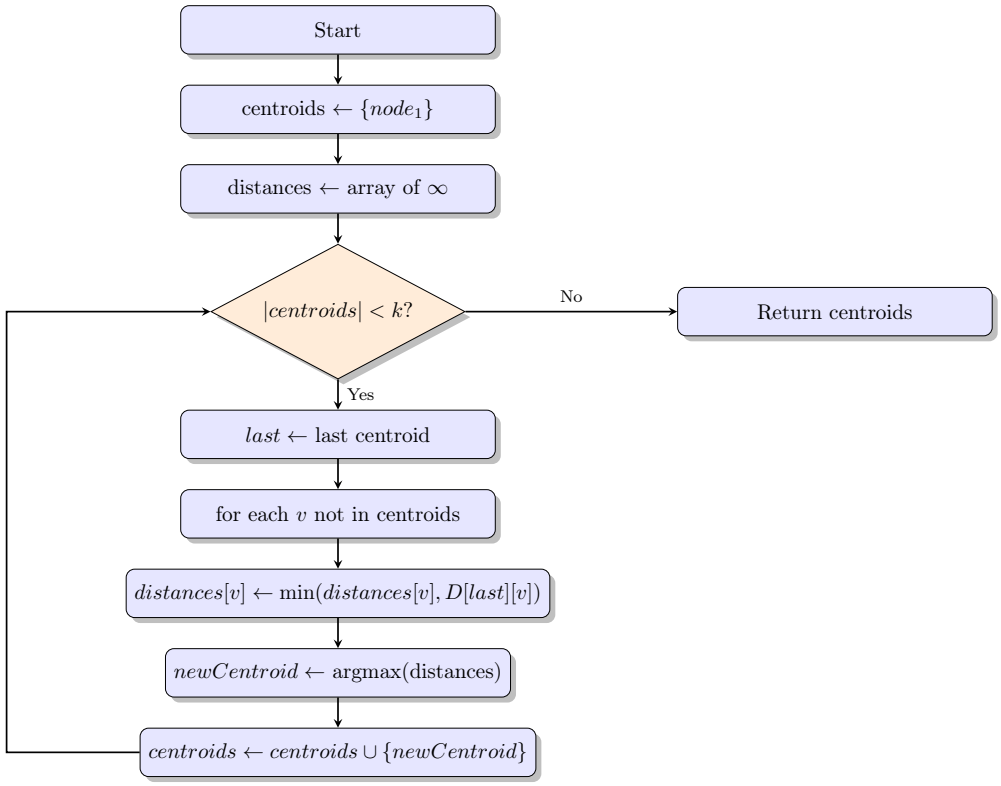

Figure 5.2: Flowchart of the centroid initialization algorithm.

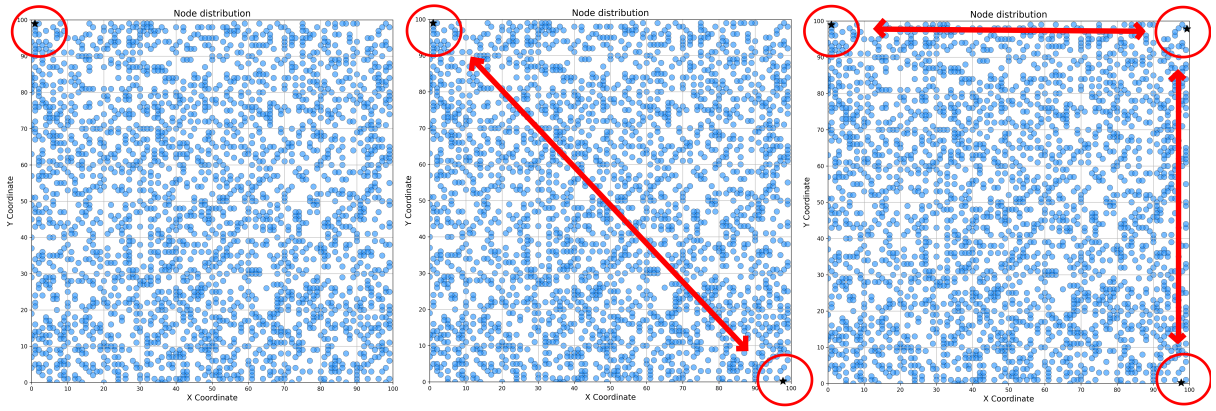

Figure 5.3: First three steps of centroid initialization

2. Balanced Cluster Population via Heap As shown in Algorithm 5.3, each cluster is associated with a priority queue (heap) that stores unassigned nodes sorted by their distance from the cluster's centroid. The algorithm proceeds iteratively, extracting the closest node from each heap and assigning it to the corresponding cluster, provided the node has not already been assigned.

To ensure balance, each cluster is capped at a predefined size, computed as n/k. Once a

node is assigned, it is marked and ignored in subsequent iterations.

- **Heap building:** $O(k \cdot n \log n)$ (each of the k heaps receives $O(n)$ insertions).
- **Population loop:** $O(k \cdot n \log n)$ in the worst case (each heap may require multiple pops).

Algorithm 5.3 Cluster Population

Require: Distance matrix D, cluster set $\{C_i\}_{i=0}^{k-1}$, centroids $\{c_i\}$, target size `clusterSize`
Ensure: Each C_i populated up to `clusterSize`

Build heaps:
for $i = 0$ to $k - 1$ **do** ▷ for each of k clusters
    initialize empty min-heap H_i ▷ key = distance to c_i
    for each node v with $assigned[v] = false$ **do**
        insert $(D[v, c_i],\ v)$ into H_i

Populate clusters:
for $i = 0$ to $k - 1$ **do**
    while $|C_i| <$ `clusterSize` **do**
        repeat
            $(distance, node) \leftarrow$ extract-min from H_i ▷ $O(\log n)$
        until $assigned[node] = false$
        add v to C_i; set $assigned[node] \leftarrow true$

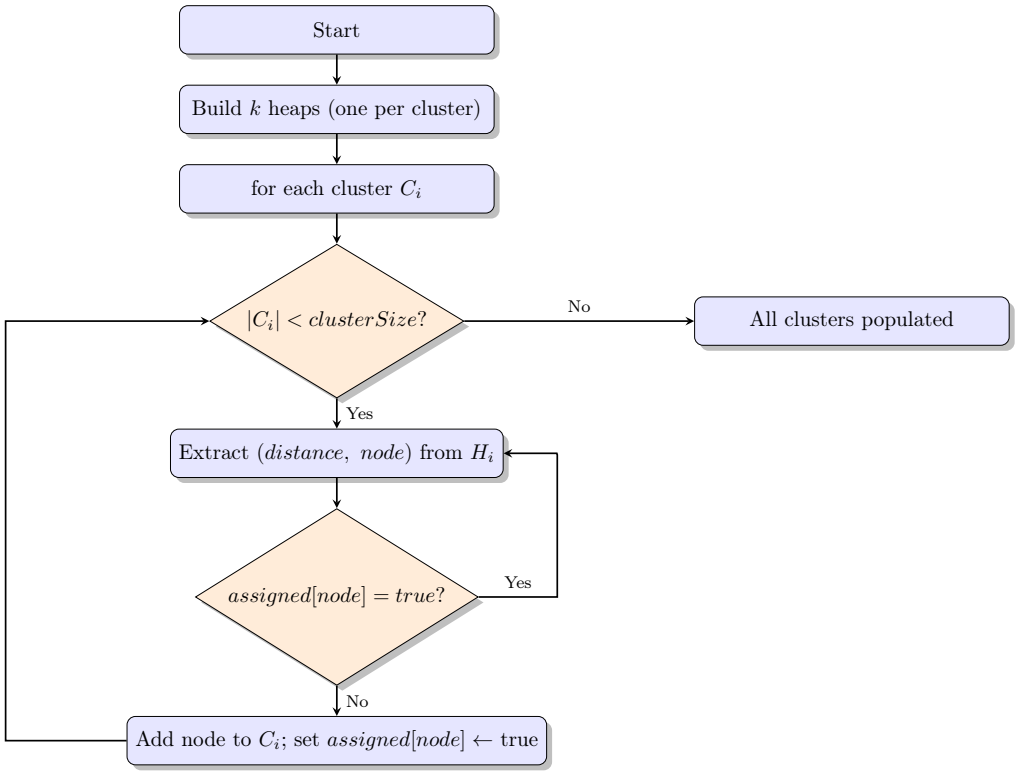

Figure 5.4: Flowchart of the balanced cluster population via heap-based assignment.

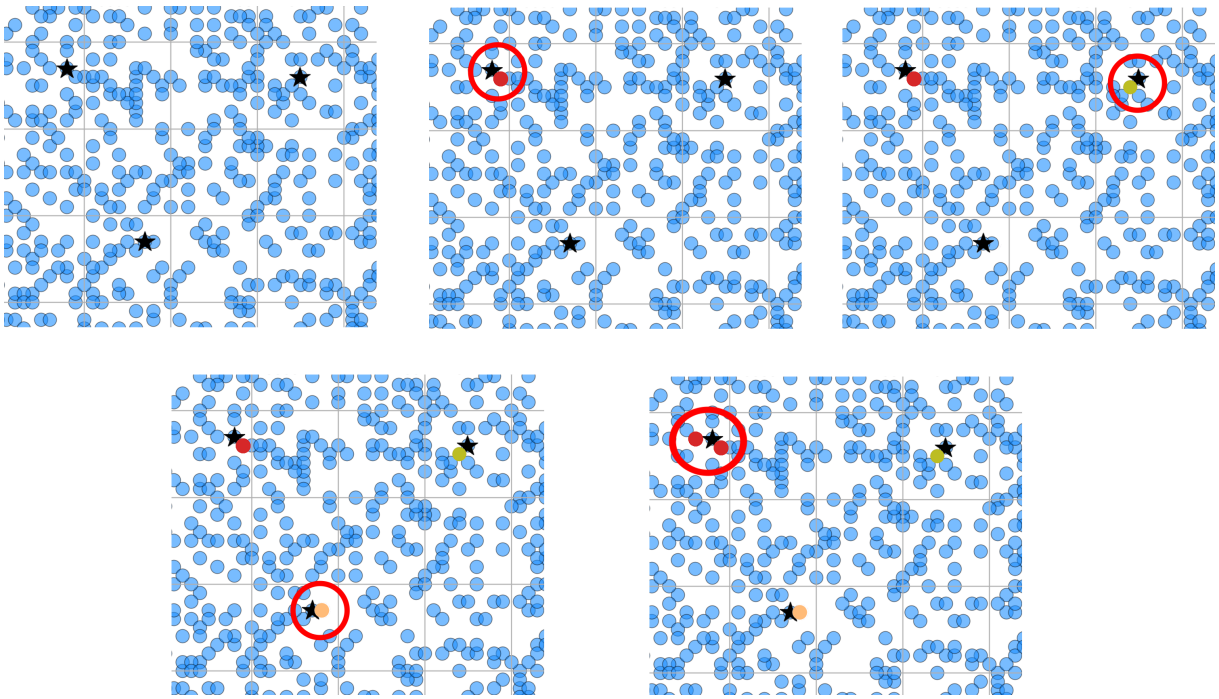

Figure 5.5: First five steps of cluster population

3. Shared Node Enforcement Algorithm 5.4 describes the post-processing stage that guarantees the required number of shared nodes per cluster. If a cluster lacks enough shared nodes, nodes are borrowed from other clusters using a heap-based proximity strategy, while ensuring the originating cluster retains its required number of exclusive nodes.

- Only non-shared nodes are eligible.
- The source cluster must retain at least the minimum number of exclusive nodes after the transfer.
- Among eligible candidates, the node closest to the target cluster's centroid is selected (again using a heap).
- For each cluster, a dedicated min-heap is constructed containing all eligible nodes from the other clusters, prioritized by their distance to the current cluster's centroid.

This stage is computationally intensive due to repeated heap constructions and validations across clusters.

- **Worst-case complexity:** $O(k \cdot n \log n)$ due to repeated scanning and heap manipulations.

Algorithm 5.4 Shared Node Enforcement

Require: Distance matrix D, cluster set *clusters*, centroids *centroids*, *minShared*, *minExclusive*
Ensure: Clusters with shared nodes

for each target cluster C_i **do**
  while $|sharedNodes(C_i)| < minShared$ **do**
    $Q \leftarrow$ empty min-heap
    for each cluster $C_j \neq C_i$ **do**
      $E \leftarrow$ exclusiveNodes(C_j)
      if $|E| > minExclusive$ **then**
        for each $v \in E$ **do**
          $d \leftarrow D[v][centroids[i]]$
          Insert (d, v) into Q
    if Q is empty **then**
      error "Not enough candidates to satisfy sharing constraint"
    $(_, v) \leftarrow$ extract-min from Q
    Add v to C_i
    Mark v as shared between clusters
function SHAREDNODES(C_i)
  return $\{v \in C_i \mid v$ is shared$\}$
function EXCLUSIVENODES(C_j)
  return $\{v \in C_j \mid v$ is not shared$\}$

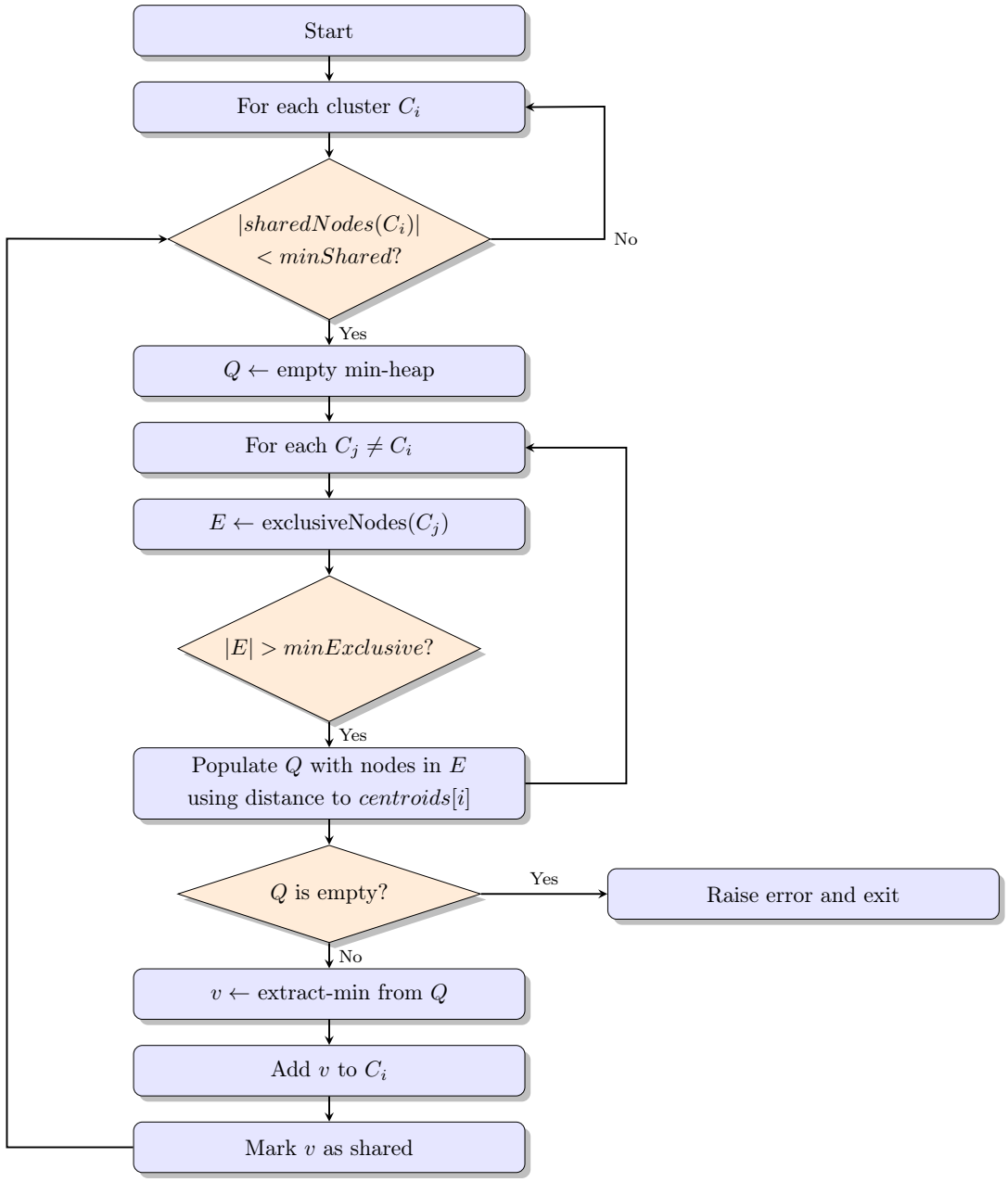

Figure 5.6: Flowchart of the Shared Node Enforcement algorithm

4. Result Structuring Finally, the algorithm constructs a summary for each cluster that includes:

- The full list of assigned nodes.
- A subset of exclusive nodes (not appearing as shared elsewhere).
- A subset of shared nodes (appearing in multiple clusters).

This is accomplished via linear scans and dictionary lookups with overall complexity $O(n)$.

Total Time Complexity Summing the above stages, the total complexity is dominated by the two heap-intensive phases:

$$T(n, k) = O(k \cdot n \log n)$$

This represents a trade-off between computational cost and the algorithm's ability to produce spatially compact clusters that strictly satisfy structural constraints.

Advantages Compared to probabilistic algorithms like SLPA:

- The results are deterministic and reproducible.
- Hard constraints on shared and exclusive nodes are enforced.
- The output is directly interpretable with labeled clusters and node roles.

This method proves especially beneficial in resource-constrained environments like edge computing, where predictable and constraint-aware clustering is critical for optimal request routing and load distribution.

5.3. Heap-Based Resource-Aware Clustering with Post-Balancing

To provide a robust clustering mechanism for NEPTUNE's dynamic edge environment, I developed an improved version of our resource-aware clustering algorithm. This new approach integrates heap-based proximity assignments[8] with a final rebalancing stage aimed at mitigating residual resource asymmetries after the initial greedy construction. It simultaneously enforces a hard cap on cluster size and achieves near-optimal resource fairness.

5.3.1. Main Idea

This algorithm addresses the cluster saturation problem in two complementary ways. First, the heap-based assignment ensures compact clusters that remain within hard bounds on both size and resource usage. Second, the final rebalancing phase actively redistributes excess resource concentration by relocating outlier nodes. Together, these strategies prevent any single cluster from becoming a bottleneck and allow the system to maintain stability under fluctuating load conditions, which is essential for edge deployments.

5.3.2. SLPA Baseline Analysis

SLPA (Speaker-listener Label Propagation Algorithm)[18, 22] partitions nodes into communities through probabilistic label propagation, but it lacks any mechanism for considering resource capacities during clustering. While Neptune adapts SLPA to fix the size of each community, this does not translate into fair resource allocation, as all nodes are treated equally regardless of their computational capabilities.

This limitation makes SLPA incompatible with scenarios that require balanced resource utilization. Our proposed resource-balanced clustering algorithm overcomes this by assigning nodes to communities based on their resource availability, and iteratively rebalancing the load to maintain uniform capacity distribution. This approach ensures that clusters are not only size-constrained but also resource-aware.

5.3.3. Novel Algorithm

Algorithm 5.5 Main Resource-Aware Clustering Algorithm

Require: Distance matrix D, resource array R, max cluster size M, tolerance τ, number of iterations T

Ensure: Set of clusters with balanced node count and resource usage

1: $n \leftarrow$ number of nodes
2: $k \leftarrow \max(2, \lceil n/M \rceil)$
3: Compute total and average resources: $\bar{R} \leftarrow \sum R_i/k$
4: *centroids* $\leftarrow$ InitializeCentroids(D, k)
5: *clusters* $\leftarrow$ HeapAssignNodes(D, *centroids*, R, M, $\bar{R}$)
6: GreedyAssignRemaining(D, *clusters*, *centroids*, R, M)
7: RebalanceResources(D, *clusters*, *centroids*, R, M, $\bar{R}$, T, τ)
8: **return** *clusters*

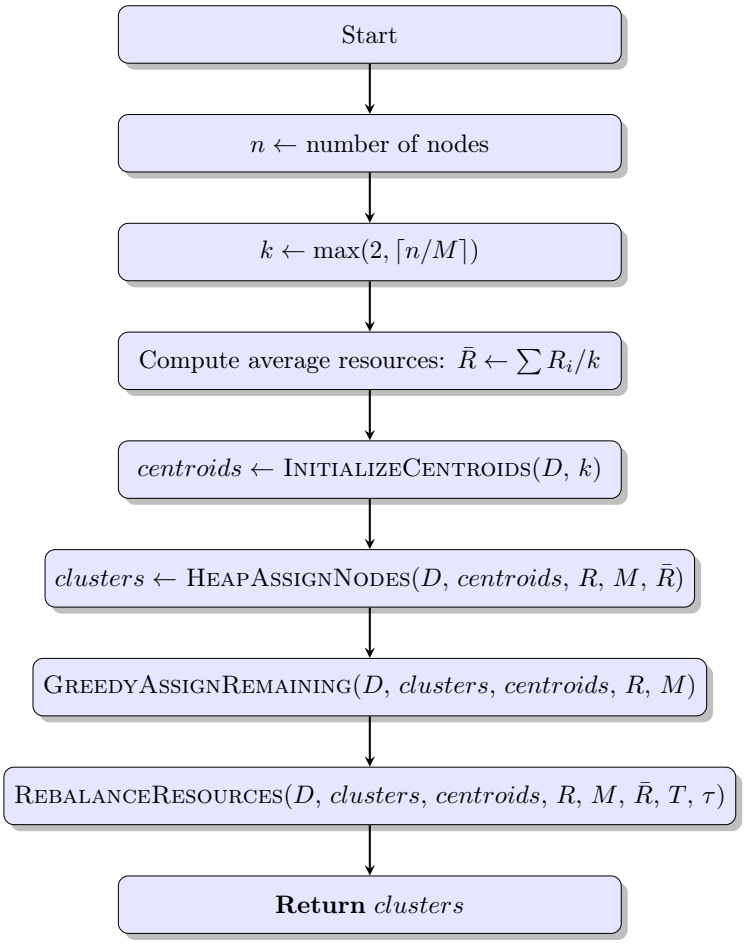

Figure 5.7: Flowchart of the Main Resource-Aware Clustering Algorithm

1. Input Parameters and Objectives The algorithm receives as input:

- A pairwise distance matrix representing node proximity.
- A list of resource values associated with each node (e.g., CPU units).
- A maximum node threshold per cluster (M).
- Optional: Number of balancing iterations (T) and tolerance factor (τ), which defaults to $T = 50$, $\tau = 1.1$.

Its objectives are:

- Generate clusters that are spatially compact and approximately equal in total resource availability.
- Ensure no cluster exceeds M nodes.
- Allow fine-tuned balancing through limited node migration between clusters.

2. Centroid Initialization Initial centroids are selected using a farthest-point greedy strategy:

1. A first node is selected randomly.
2. Each new centroid maximizes its distance to the closest existing centroid.

Complexity: $O(k \cdot n)$

Algorithm 5.6 Centroid Initialization

Require: Distance matrix D, number of clusters k
Ensure: Set of k centroids

$centroids \leftarrow \{node_1\}$ ▷ Select first node
$distances \leftarrow$ array of n values ∞
while $|centroids| < k$ **do**
    $last \leftarrow$ last element of $centroids$
    for each node v not in $centroids$ **do**
        $distances[v] \leftarrow \min(distances[v], D[last][v])$
    $newCentroid \leftarrow$ node with maximum $distances[v]$
    $centroids \leftarrow centroids \cup \{newCentroid\}$
return $centroids$

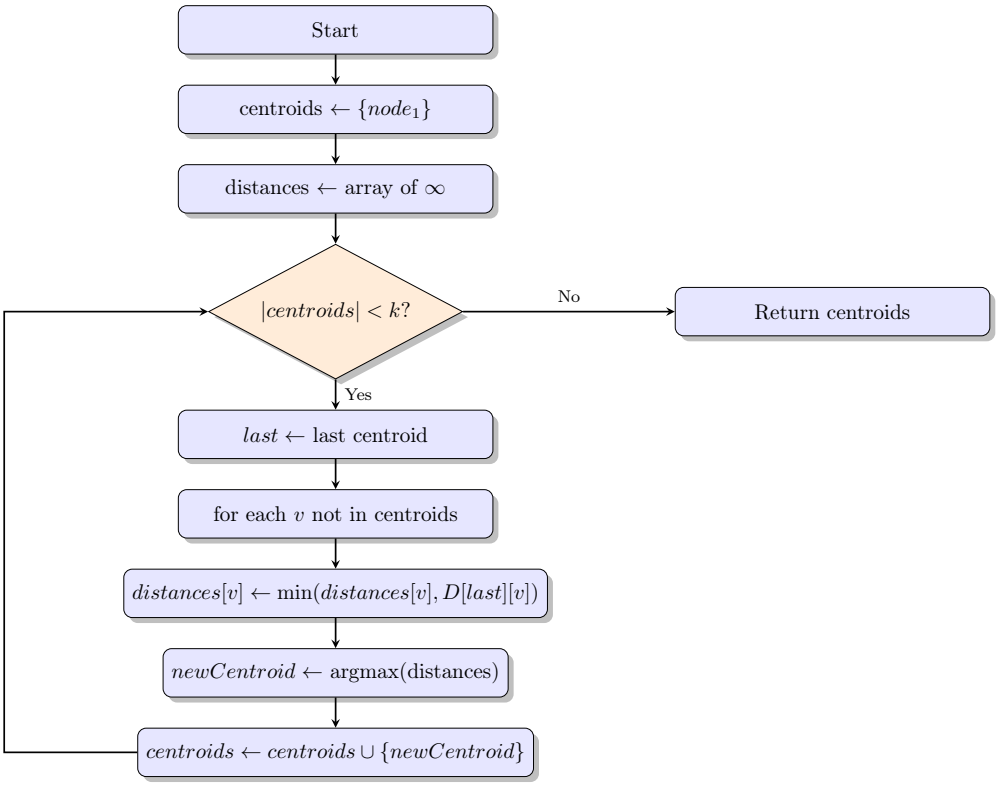

Figure 5.8: Flowchart of the centroid initialization algorithm.

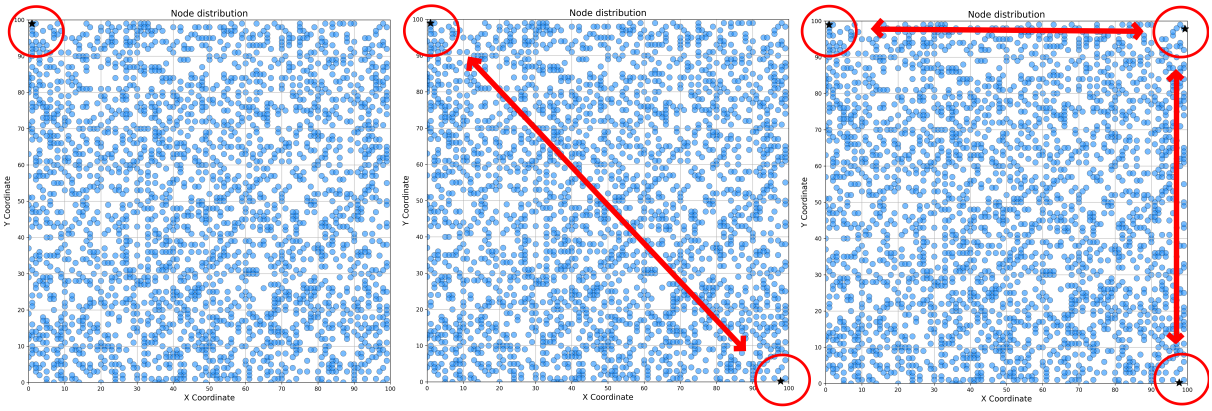

Figure 5.9: Progression of centroid initialization

3. Heap-Based Node Assignment Each centroid is associated with a heap of candidate nodes sorted by proximity. Nodes are assigned iteratively to the closest cluster, respecting both the maximum node limit M and an upper bound of $1.1 \times$ avg_resources per cluster.

Complexity:

- Heap construction: $O(k \cdot n \log n)$

- Assignment: $O(k \cdot n \log n)$

Algorithm 5.7 Heap-Based Node Assignment

Require: Distance matrix D, centroids *centroids*, resources R, max cluster size M, avg resources $\bar{R}$, tolerance τ

Ensure: Initial clusters

```
Initialize k empty clusters, one for each centroid
for each cluster C_i do
    Add centroid to C_i, mark assigned
    Build min-heap of unassigned nodes by distance to centroid
while There are still nodes to be assigned do
    for each cluster C_i do
        v ← first unassigned node in the heap for C_i
        if v exists and |C_i| < M and R(C_i) + R_v ≤ τ · R̄ then
            Assign v to C_i, mark as assigned
return clusters
```

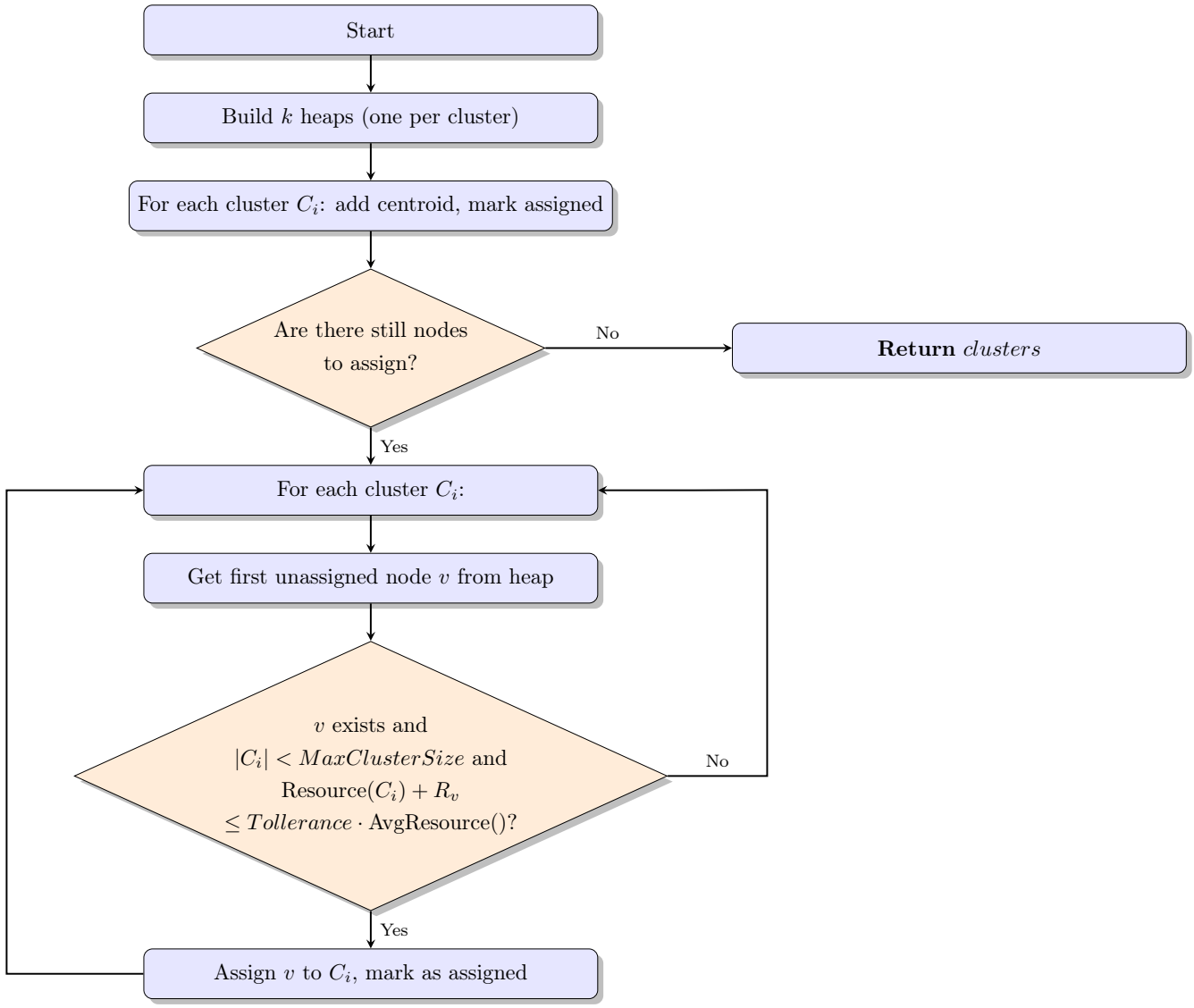

Figure 5.10: Flowchart of the Heap-Based Node Assignment algorithm

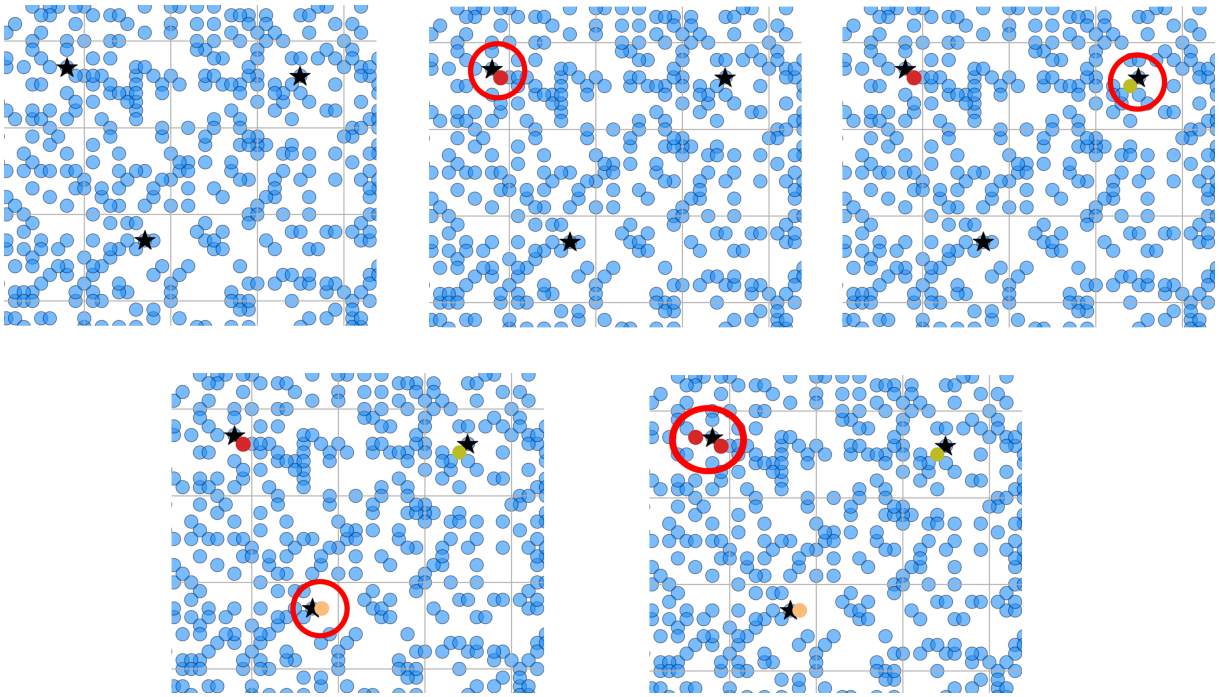

Figure 5.11: First 5 step of cluster population

4. Greedy Fallback Assignment Nodes left unassigned due to heap constraints are greedily assigned to the nearest cluster with available capacity.

Complexity: $O(n \cdot k)$

Algorithm 5.8 Greedy Fallback Assignment

Require: Distance matrix D, clusters *clusters*, centroids *centroids*, resources R, max cluster size M

Ensure: All nodes are assigned

1: **for** each unassigned node v **do**
2: Compute distances to each centroid
3: Assign v to nearest cluster with size $< M$, update resource totals

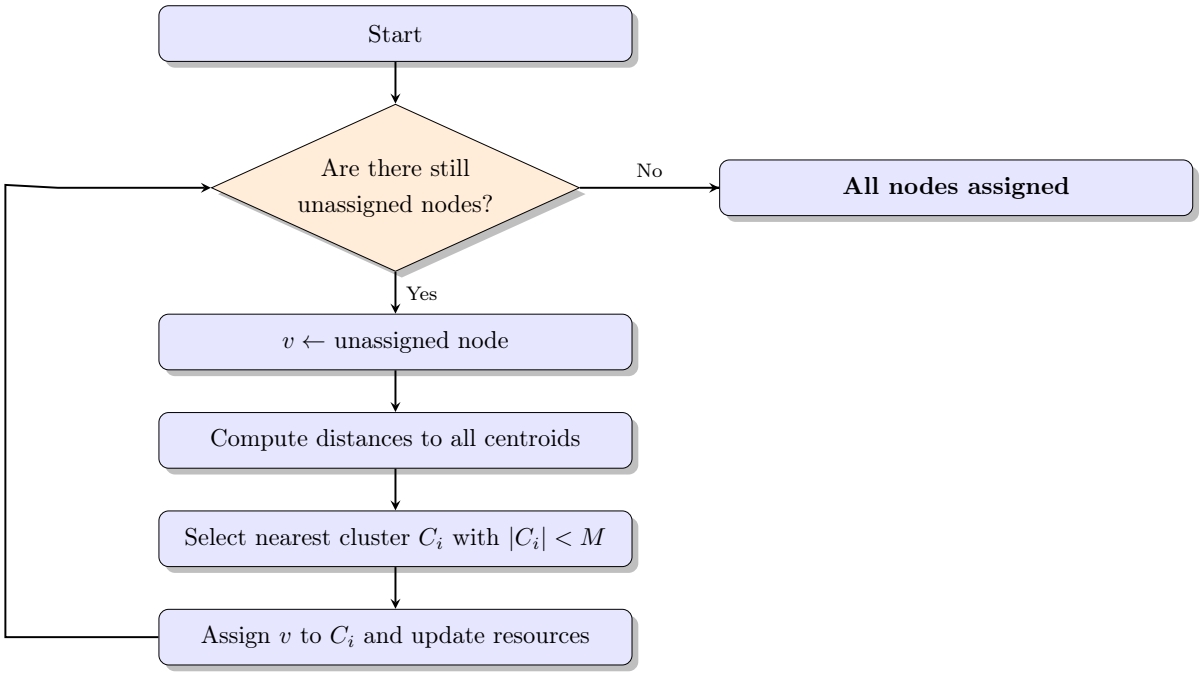

Figure 5.12: Flowchart of the Greedy Fallback Assignment algorithm

5. Final Resource Balancing (T Iterations) The final stage of the algorithm performs T iterations of resource rebalancing:

- In each cluster, the node farthest from the centroid (excluding the centroid itself) is selected as a candidate for migration.
- This node is then reassigned to the nearest eligible cluster (i.e., with lower total resources and fewer than $\tau \cdot M$ nodes).
- Resource and node counters are updated accordingly.

This step aims to reduce variance in total resource allocation without disrupting spatial compactness.

Complexity: $O(T \cdot k \cdot \log k)$ in practice, with each step involving distance evaluations and list updates.

Figure 5.13 shows three steps of a single iteration of the resource balancing algorithm. In this example, three clusters are depicted, initially containing 850, 700, and 900 units of resource respectively.

At each step, the algorithm selects one cluster (in this case, the purple one with 900 units), identifies the node that is furthest from its centroid, and attempts to transfer it to a suitable neighboring cluster. The destination cluster must have fewer resources, not

exceed a resource threshold after the transfer, and be spatially close to the selected node.

In the example, the purple cluster has 900 units of resource, and the brown cluster has 700. Since the brown cluster satisfies the constraints, the furthest node (circled) is moved from the purple to the brown cluster. Assuming this node contributes 10 units of resource, the updated resource distribution becomes: 890 for the purple cluster, 710 for the brown cluster, and 850 for the cyan cluster.

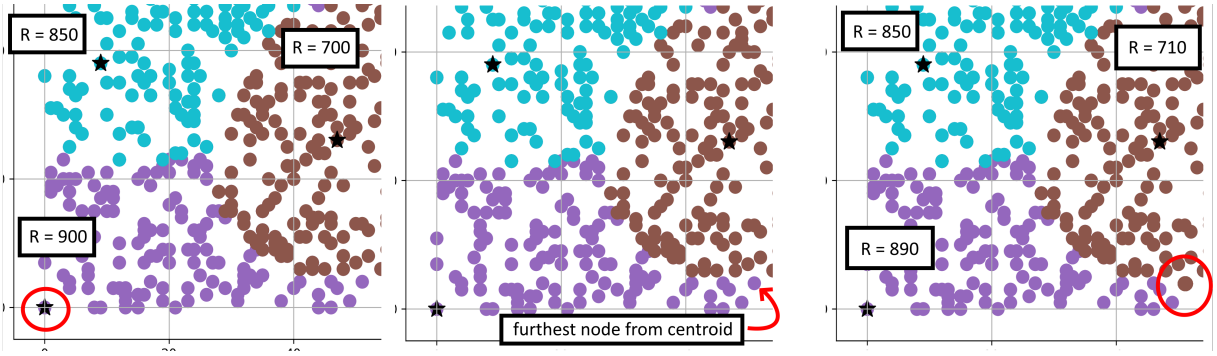

Figure 5.13: Illustration of three consecutive steps in one iteration of the resource balancing algorithm.

Algorithm 5.9 Final Resource Balancing ($\boldsymbol{T}$ Iterations)

Require: Distance matrix D, clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$, centroids centroid_i for each C_i, resource vector R, max cluster size M, average resource per cluster $\bar{R}$, number of iterations T, tolerance τ

Ensure: Clusters with more balanced resource distribution

for $t = 1$ to T **do**
  for each cluster C_i **do**
    if $|C_i| \leq 1$ **then**
      continue ▷ Skip if cluster too small to move from
    Let $v \in C_i$ be the node farthest from centroid_i
    Initialize empty candidate list
    for each cluster C_j with $j \neq i$ **do**
      if $|C_j| < \tau \cdot M$ **and** $R(C_j) < R(C_i)$ **and** $R(C_j) + R_v \leq \tau \cdot \bar{R}$ **then**
        Add $(D[v, \text{centroid}_j], j)$ to candidate list
    if candidate list is not empty **then**
      Select C_j with minimal $D[v, \text{centroid}_j]$
      Move node v from C_i to C_j
      Update $R(C_i), R(C_j)$ and cluster sizes accordingly

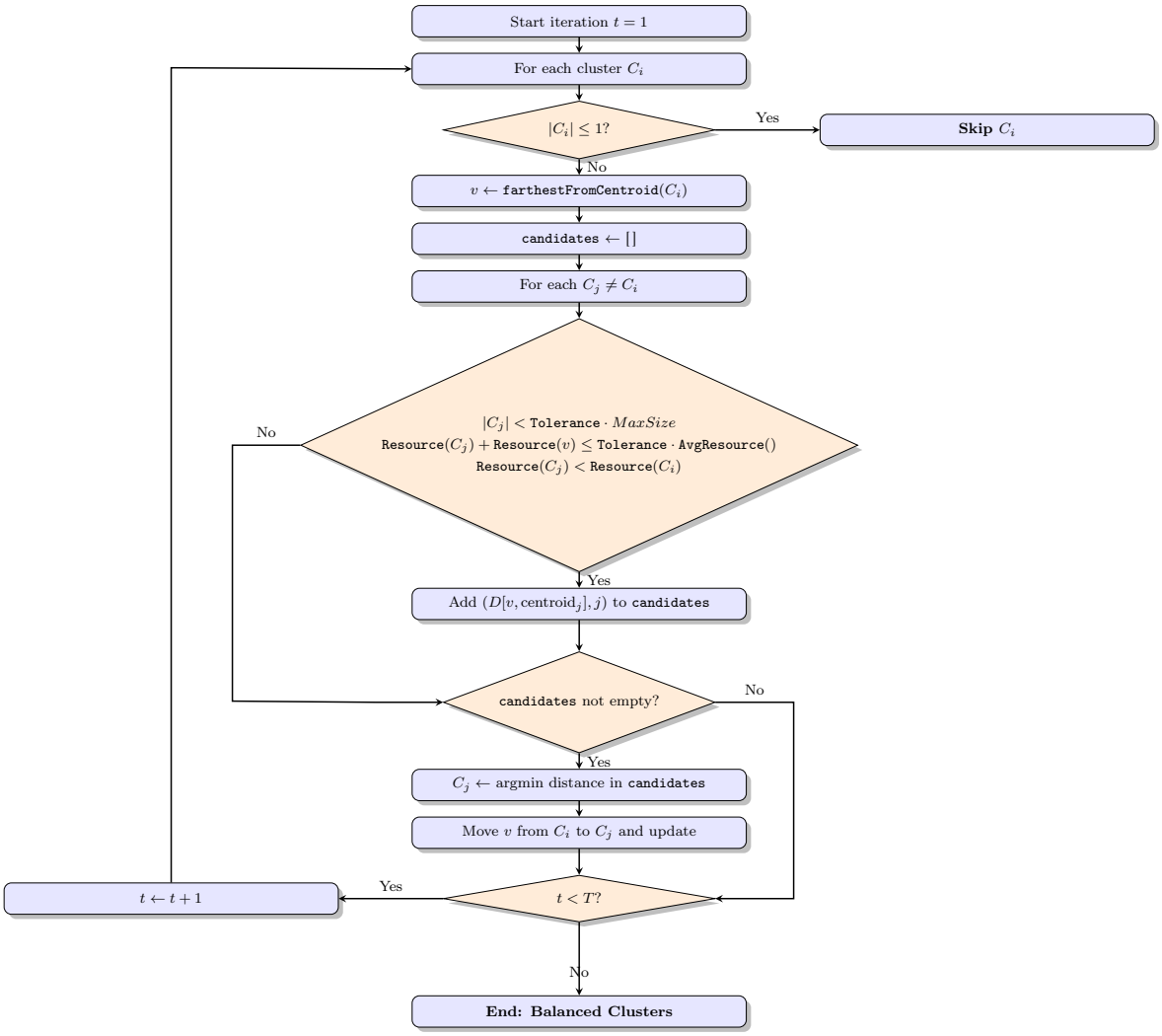

Figure 5.14: Flowchart of the Final Resource Balancing algorithm

Output Each cluster is represented by:

- The list of assigned nodes.
- The total aggregated resources.
- The number of nodes.

Total Time Complexity Summing all stages, I obtain:

$$T(n, k) = O(k \cdot n \log n + T \cdot k \cdot \log k)$$

For fixed T, this simplifies to:

$$T(n, k) = O(k \cdot n \log n)$$

Advantages This enhanced algorithm provides:

- Strong initial spatial coherence through heap-based assignments.
- Tight control over cluster size and resource upper bounds.
- A tunable rebalancing phase that smooths out suboptimalities.
- Full node coverage and graceful handling of constraint conflicts.

Overall, this strategy delivers high-quality clusterings suitable for NEPTUNE's multi-objective scheduling pipeline, especially under realistic, resource-constrained MEC conditions.

5.4. EF-TTC Function Placement

The third research direction focuses on enhancing NEPTUNE's function placement module [2] by integrating a novel heuristic algorithm, named EF-TTC (Edge Function Topology-Traffic Cycles) [14]. The primary goal is to provide an efficient and scalable alternative to the existing Mixed-Integer Programming (MIP)-based solver[21], particularly in large-scale and heterogeneous edge environments.

5.4.1. MIP Baseline Analysis

The integration process began with a comprehensive analysis of the existing MIP-based placement module. This module is designed to optimize function placement based on one of three objectives:

- **Minimize Delay**: Reducing the total network delay induced by function invocation across nodes.
- **Minimize Utilization**: Minimizing the number of active nodes and their associated resource usage costs.
- **Minimize Delay and Utilization**: A hybrid strategy balancing both delay and utilization through a weighted sum.

In addition to objective alignment, several strict constraints needed to be maintained:

- Memory and CPU capacity limits per node.

- One instance per function per node.
- Full request routing: all workload demands must be redirected to valid function instances.
- Budget limits on node activations.

5.4.2. Constraint Satisfaction and Variable Semantics

To ensure that EF-TTC [14] preserves the correctness and feasibility of the solution space originally enforced in NEPTUNE's MIP-based approach [2], several constraints are respected at each allocation step. These constraints guarantee correct resource usage, budget adherence, and complete routing of requests.

Main variables:

- $x[i, f, j]$: Fraction of requests to function f from node i that are routed to node j.
- $c[f, j]$: Boolean variable that is 1 if function f is deployed on node j.
- $n[j]$: Boolean variable that is 1 if node j is activated in the allocation.

Enforced constraints:

1. **Memory constraint:** The total memory used by functions on a node cannot exceed its memory capacity.
2. **CPU constraint:** Each node must have sufficient cores to serve all the requests routed to it.
3. **Budget constraint:** The sum of the costs of the active nodes must not exceed the budget limit of the community.
4. **Activation consistency:** A node is marked as active if and only if at least one function is deployed on it.
5. **Deployment consistency:** A function is considered deployed on a node if and only if some requests are routed to it.
6. **Routing completeness:** All requests originating from any node i for a given function f must be entirely routed to some destination.

Constraint formulas summary:

Constraint	Formula
Memory usage	$\sum_{f} c[f,j] \cdot \text{mem}(f) \leq \text{mem}(j)$
CPU usage	$\sum_{i,f} x[i,f,j] \cdot \text{workload}[f,i] \cdot \text{core}[f,j] \leq \text{cores}(j)$
Budget	$\sum_{j} n[j] \cdot \text{cost}(j) \leq \text{budget}$
Node activation	$\sum_{f} c[f,j] > 0 \Rightarrow n[j] = 1$
Routing $\Rightarrow$ Deployment	$\sum_{i} x[i,f,j] > 0 \Rightarrow c[f,j] = 1$
Routing completeness	$\sum_{j} x[i,f,j] = 1 \quad \forall i,f$ such that $\text{workload}[f,i] > 0$

Table 5.1: Formal definition of EF-TTC constraints (non-GPU version).

5.4.3. Adapting EF-TTC to the MIP Framework

To preserve compatibility with NEPTUNE's architecture [2], EF-TTC [14] was designed to mirror the original solver's input-output structure, including the format of function placement matrices and request routing variables.

The EF-TTC approach builds on the principle of preference graphs. Two sets of preferences are computed:

- For each function f, a ranked list of nodes j based on their estimated contribution to the objective function.
- For each node j, a ranked list of functions f according to local utility or delay gains.

These rankings are computed via a set of *local score functions* which approximate each global objective

5.4.4. Objective Functions and Local Scoring in EF-TTC

The original NEPTUNE MIP-based placement [2] supported three distinct objective functions:

1. **Minimize Delay:** reduce the total latency of serving requests.
2. **Minimize Utilization:** minimize the number of active nodes used.

3. **Minimize Delay and Utilization:** a weighted combination of the two above, controlled via parameter $\alpha \in [0, 1]$.

To replicate the same optimization goals within the EF-TTC algorithm [14], each global objective function was translated into a local numeric scoring function. These local functions evaluate the impact of assigning a specific function to a given node within the current context, providing a scalar value that approximates how the allocation affects the overall objective. This allows the EF-TTC heuristic to make placement decisions that are locally guided yet aligned with the global optimization target.

Global Objective Function for MIP processing

1. Network Delay Objective

The total delay is computed as the sum of workloads routed through the network, weighted by their associated delays:

$$\text{Score} = \sum_{i,f,j} x_{i,f,j} \cdot d_{i,j} \cdot w_{f,i}$$

where $x_{i,f,j}$ is the routing decision, $d_{i,j}$ the delay, and $w_{f,i}$ the workload.

2. Node Utilization Objective

The number of active nodes is computed as:

$$\text{Score} = \sum_{j} n_j$$

3. Combined Delay and Utilization

A normalized combination of the two objectives:

$$\text{Score} = \alpha \cdot \frac{\sum_j n_j}{|N|} + (1 - \alpha) \cdot \frac{\text{delay_score}}{\text{max_workload_delay}}$$

where:

- α: trade-off parameter between utilization and delay,
- `max_workload_delay`: normalization constant for delay.

Local Score Approximation for TTC Graph Construction

To ensure efficiency and greedy behavior in EF-TTC, each function-node pair (f, j) is assigned a local score:

MinUtilization:

$$s_{f,j} = \frac{c_j}{1 + u_j} \cdot b_{f,j}$$

encourages reuse of active nodes; c_j is node cost, u_j its utilization, and $b_{f,j}$ the warm-start bonus.

MinDelay:

$$s_{f,j} = \left(\sum_i d_{i,j} \cdot w_{f,i} \right) \cdot b_{f,j}$$

estimates the delay if f is placed on j.

MinDelay+Utilization:

$$s_{f,j} = \left(\alpha \cdot \frac{c_j}{1 + u_j} + (1 - \alpha) \cdot \sum_i d_{i,j} \cdot w_{f,i} \right) \cdot b_{f,j}$$

balances delay and node usage.

Note:
u_j is the number of functions currently assigned to node j,
$b_{f,j} = 0.5$ if f was previously on j (warm start), otherwise 1.0.

These local scores are used during the TTC graph construction phase, where each function f chooses its most preferred node j and vice versa. This localized heuristic improves runtime while still aligning with the global objective.

The following table summarizes the three main optimization strategies supported by EF-TTC, along with the corresponding global objective functions and local scoring formulas used for preference graph construction.

Objective	Global Objective Function	Local Score (per f, j)
Minimize Delay	$\sum_{i,f,j} x[i,f,j] \cdot \text{delay}[i,j] \cdot \text{workload}[f,i]$	$\left(\sum_i \text{delay}[i,j] \cdot \text{workload}[f,i] \right) \cdot \text{warm_bonus}$
Minimize Utilization	$\sum_j n[j]$	$\frac{\text{cost}(j)}{1 + \text{util}(j)} \cdot \text{warm_bonus}$
Minimize Delay + Utilization	$\alpha \cdot \frac{\sum_j n[j]}{\lvert N \rvert} + (1 - \alpha) \cdot \frac{\text{delay_score}}{\text{max_workload_delay}}$	$\left(\alpha \cdot \frac{\text{cost}(j)}{1 + \text{util}(j)} + (1 - \alpha) \cdot \sum_i \text{delay}_{i,j} \cdot \text{workload}_{f,i} \right) \cdot \text{warm_bonus}(f,j)$

Table 5.2: Objective functions and corresponding local scores used for TTC-based matching.

5.4.5. Cycle Detection and TTC Application

EF-TTC [14] constructs a bipartite preference graph with functions and nodes, where edges represent top preferences. It then searches for **transfer cycles** of the form:

$$f_1 \to j_1 \to f_2 \to j_2 \to \cdots \to f_k \to j_k \to f_1$$

Each cycle identifies a potential reallocation sequence, allowing function instances to be shifted across nodes in a coordinated manner, maintaining feasibility while improving the overall score.

Upon detecting a cycle, the algorithm:

1. Verifies that adding the proposed function-to-node pairs does not violate any hard constraint.
2. If valid, updates the placement and request routing structures accordingly.
3. If invalid, the cycle is discarded and recorded in a blacklist to prevent re-evaluation.

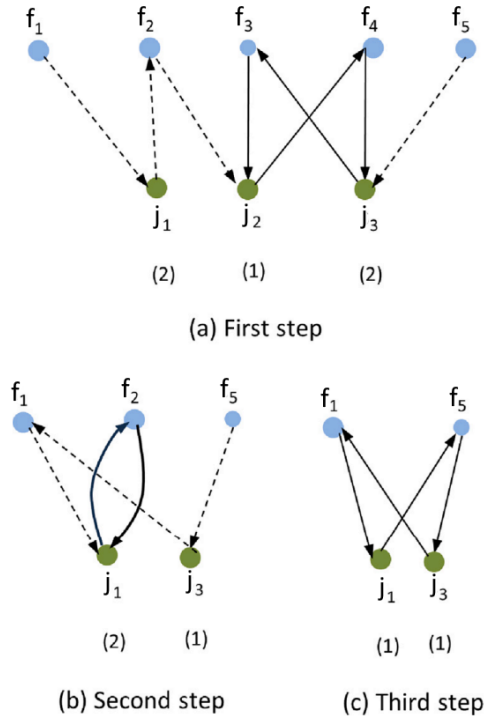

Figure 5.15: An example of how the TTC mechanism works from *An efficient mechanism for function scheduling and placement in function as a service edge environment* [14].

5.4.6. Convergence Strategy

The iterative process continues as long as there are functions that could:

- Be allocated to a new node with potential score improvement, or
- Be assigned while maintaining all constraints.

A function f is removed from the *remaining function set* when no further beneficial allocation is possible. The removal logic depends on the optimization objective:

- **MinUtilization**: Once f is successfully assigned to a valid node j, it is immediately removed from the remaining set, as reallocation would only increase utilization.
- **MinDelay** or **Combined**: f remains in the set until no unvisited node j can improve the delay score. The best candidate node is determined using a delay-aware heuristic:
  1. Filter out nodes already assigned to f or marked as invalid.
  2. For each source node i with non-zero workload toward f, identify the node(s) j with minimal delay $d_{i,j}$.
  3. Collect all such best nodes across sources and compute their aggregate delay score: $\text{score}_j = \sum_i w_{f,i} \cdot d_{i,j}$.
  4. Select the node j^* with the lowest score_j. If the score is significantly better than 0, f remains in the set; otherwise, it is removed.

Nodes that reach their full capacity are excluded from future allocations.

This convergence mechanism ensures that EF-TTC terminates in polynomial time with a valid placement satisfying all routing and resource constraints, or concludes that no further improvement is possible.

5.4.7. EF-TTC Algorithm: Design and Complexity Analysis

The Edge Function Top Trading Cycle (EF-TTC) [14] algorithm addresses the function placement problem under memory and delay constraints by iteratively reallocating functions through trading cycles. At each iteration, the algorithm constructs a preference graph based on scoring heuristics and attempts to apply valid reassignments if a cycle is detected. The objective is to iteratively reduce delays while maintaining feasible resource usage across all nodes.

Let F be the number of functions and N the number of nodes. In the worst-case scenario,

up to $O(F \cdot N)$ iterations might be required, since each function could be re-evaluated for assignment on each node. The per-iteration steps and their associated costs are described below.

Algorithm 5.10 EF-TTC Assignment with Cycle Reallocation

Require: Functions F, Nodes N, Memory matrix M, Delay matrix Δ
Ensure: Valid assignment A

```
Initialize assignment state and invalid pair set 𝓘
while there are functions needing assignment or improvement do
    Clear score cache
    G ← BuildPreferenceGraph(F, N, 𝓘)
    𝓒 ← FindCycle(G)
    if 𝓒 is null or already tried then break
    Backup current state
    if CanAssignCycle(𝓒) then
        if CheckConstraints() then
            HandleCycle(𝓒)
        else
            Restore previous state, mark 𝓒 as invalid
    else
        Mark invalid pairs in 𝓒
```

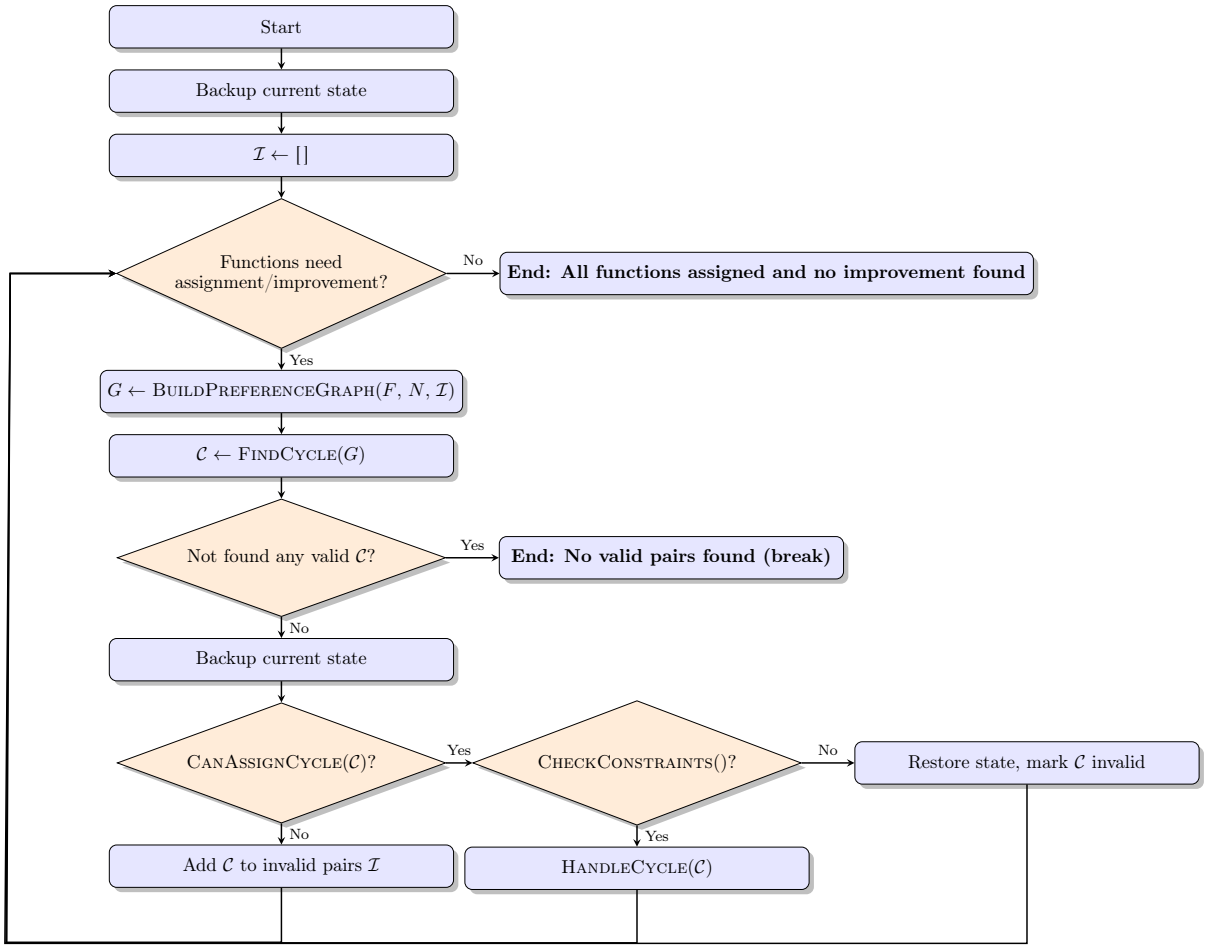

Figure 5.16: Flowchart of the EF-TTC Assignment with Cycle Reallocation

Algorithm Components and Complexity:

- **Preference Graph Construction** (BuildPreferenceGraph): Each function and node ranks its best counterpart based on scoring metrics, leading to a directed bipartite graph. Cost: $O(F \cdot N)$.

Algorithm 5.11 BuildPreferenceGraph(F, N, $\mathcal{I}$)

for each function $f \in F$ **do**
    Rank valid nodes $j \in N$ based on score
    Add edge $f \rightarrow \arg\min_j$ (best node)
for each node $j \in N$ **do**
    Rank functions $f \in F$ based on score
    Add edge $j \rightarrow \arg\min_f$ (best function)
return directed graph G

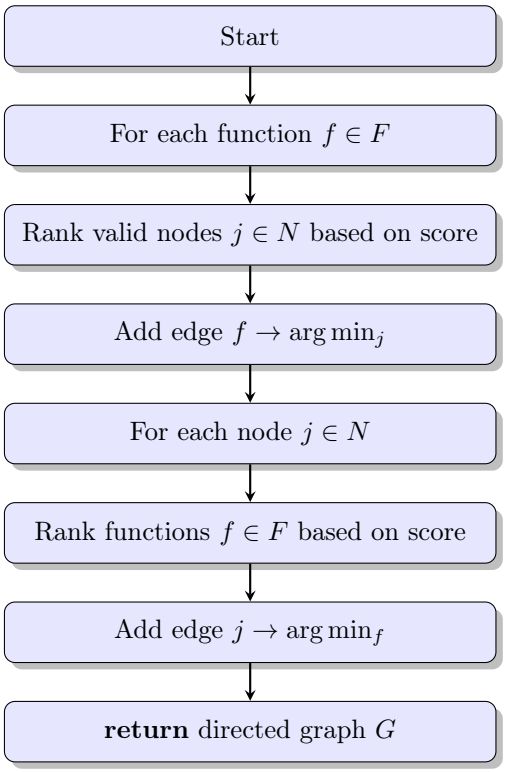

Figure 5.17: Flowchart of BUILDPREFERENCEGRAPH

- **Cycle Detection** (FINDCYCLE): Iteratively traverses the preference graph to detect a valid trading cycle between functions and nodes. Cost: $O(F \cdot N)$.

Algorithm 5.12 FINDCYCLE(G)

```
Initialize global visited set 𝒱
for each node v ∈ G do
    if v ∈ 𝒱 then continue
        Initialize empty path list P and local visited set ℒ
        current ← v
        while current ∉ ℒ and current ∈ G do
            Add current to ℒ and append to P
            current ← G[current]                    ▷ Follow preference edge
        if current ∈ ℒ then
            Extract cycle subpath from P starting at first occurrence of current
            Transform into list of function-node pairs (f, j)
            return cycle 𝒞
    𝒱 ← 𝒱 ∪ ℒ
return null
```

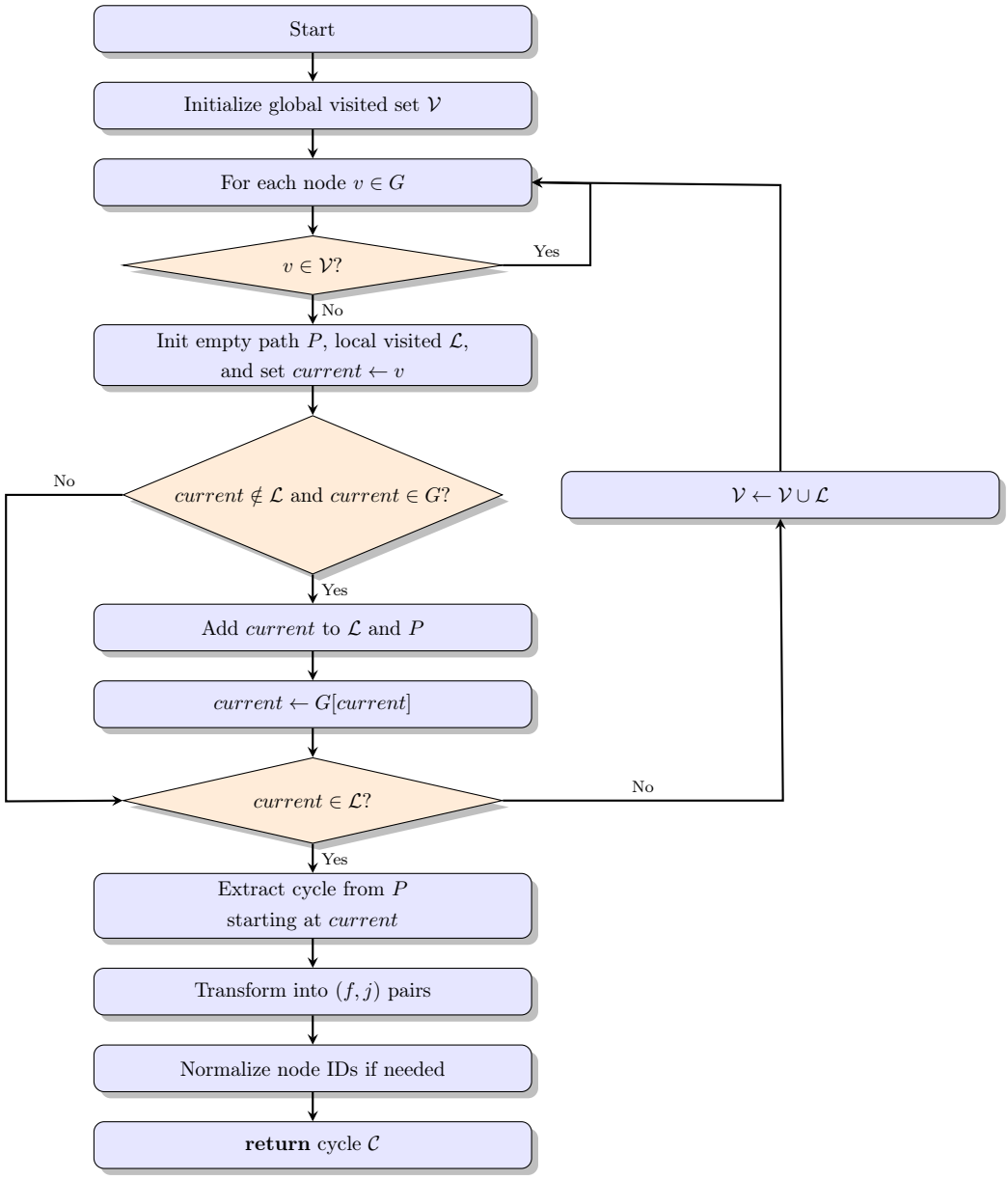

Figure 5.18: Flowchart of FindCycle(G)

- **Snapshot and Restore**: Full state backup or rollback involves memory assignments and function-routing weights. Cost: $O(F \cdot N^2)$.
- **Cycle Assignment** (CanAssignCycle): Verifies whether each function in the cycle can be placed on its designated node without violating capacity. Cost: $O(F \cdot N)$.

Algorithm 5.13 CanAssignCycle($\mathcal{C}$)

for each $(f, j) \in \mathcal{C}$ **do**
    if f can be assigned to j without violating memory **then**
        Assign f to j
        Update routing weights
    else
        Mark (f, j) as invalid
return `true if at least one assignment succeeded`

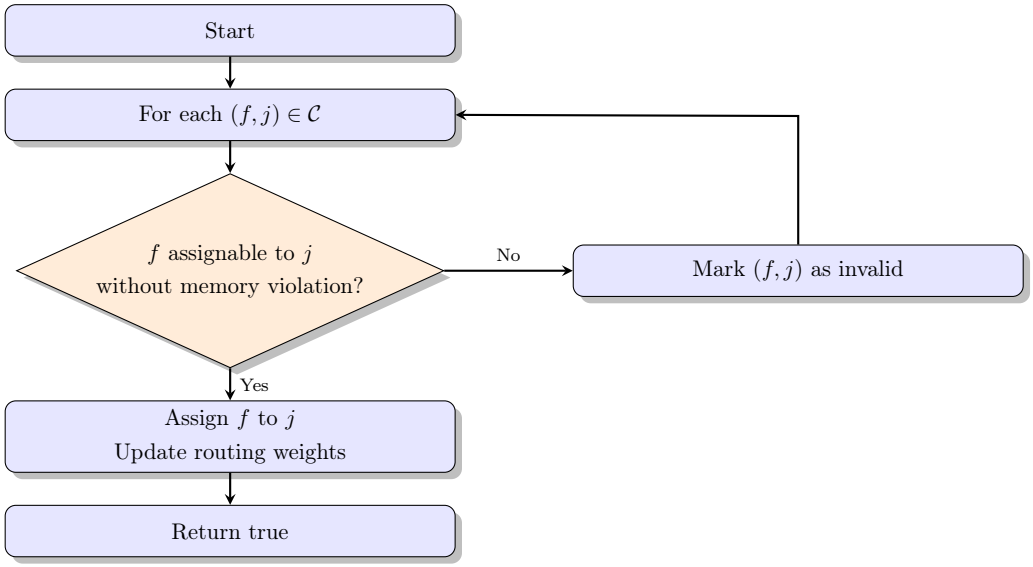

Figure 5.19: Flowchart of CanAssignCycle($\mathcal{C}$)

- **Cycle Handling** (HandleCycle): Updates node saturation, and verifies if functions can still be improved. Cost: $O(F \cdot N)$.

Algorithm 5.14 HandleCycle($\mathcal{C}$)

for each $(f, j) \in \mathcal{C}$ **do**
    if node j is full **then**
        Remove j from tje remaining nodes
    if memory overused **then**
        Restore previous state, mark as invalid
for each f in $\mathcal{C}$ **do**
    if CanImproveAssignment(f, N, A, $\mathcal{I}$) = `false` **then**
        Remove f from F

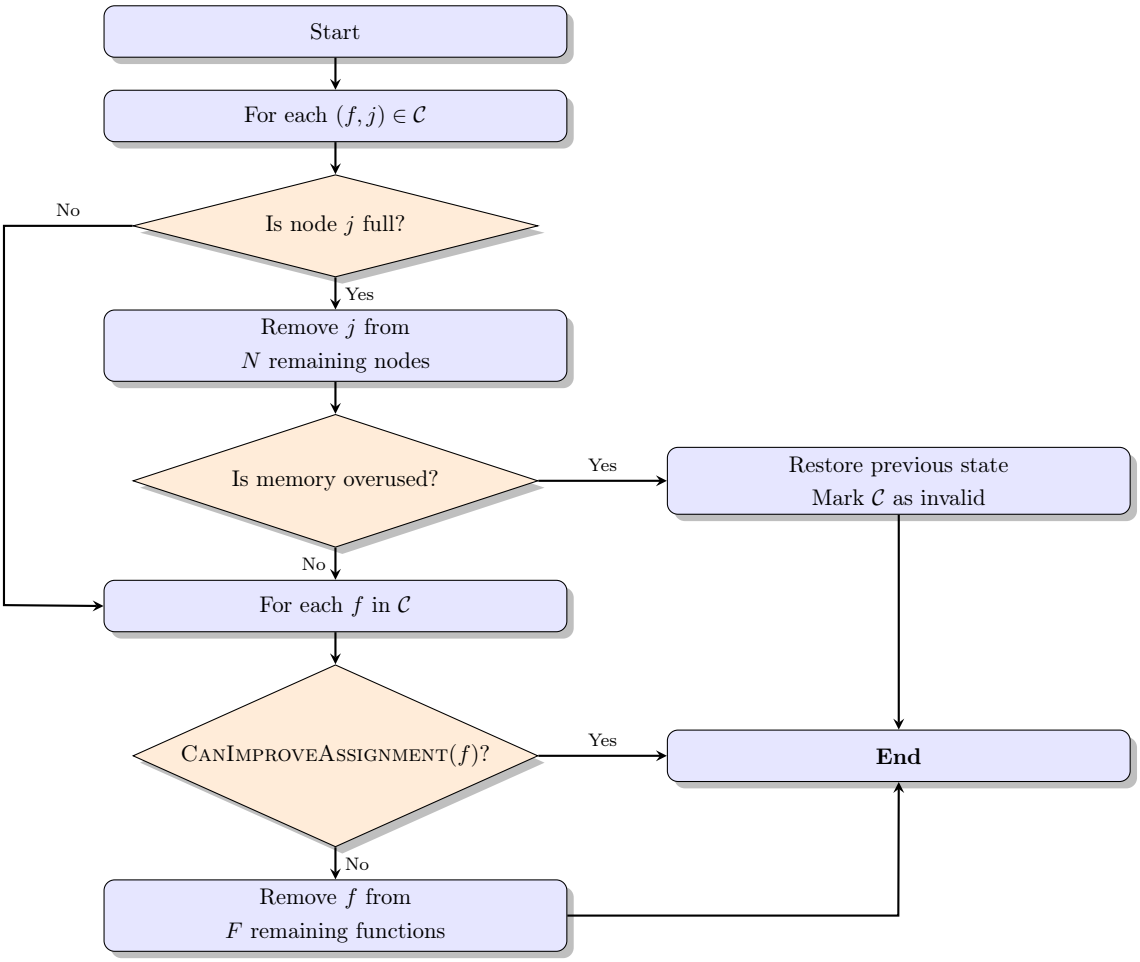

Figure 5.20: Flowchart of HandleCycle($\mathcal{C}$)

- **Improvement Check** (CanImproveAssignment): Evaluates if a function can still reduce its delay by being reassigned to a better node. Cost: $O(F \cdot N)$.

Algorithm 5.15 CanImproveAssignment(f, N, A, $\mathcal{I}$)

Let W_f be the workload vector of function f
Let D be the node delay matrix
$J_{active} \leftarrow \{j \in N \mid A(f, j) = \texttt{true}\}$
Compute $current_delay \leftarrow \sum_i W_f[i] \cdot \min_{j \in J_{active}} D[i][j]$
for each $j \in N \setminus J_{active}$ such that $(f, j) \notin \mathcal{I}$ **do**
    $D'_i \leftarrow \min(D[i][j], \min_{j' \in J_{active}} D[i][j'])$ for all i
    $new_delay \leftarrow \sum_i W_f[i] \cdot D'_i$
    if $current_delay > new_delay$ **then**
        return `true`
return `false`

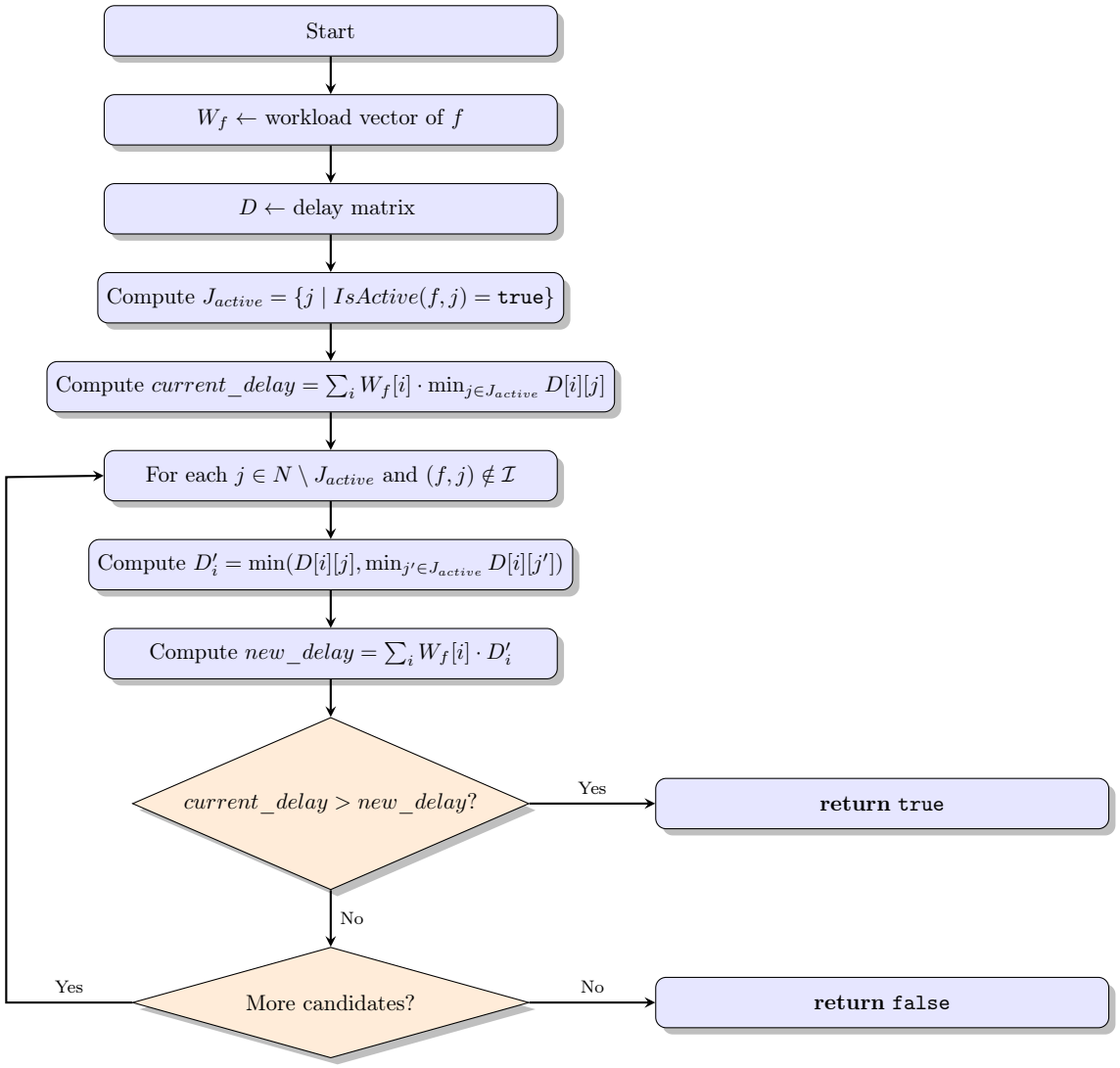

Figure 5.21: Optimized flowchart of CanImproveAssignment(f, N, A, $\mathcal{I}$)

Total Complexity: The overall worst-case complexity is given by:

$$O(F \cdot N) \times O(F \cdot N^2) = O(F^2 \cdot N^3)$$

This reflects the maximum number of iterations multiplied by the most expensive step per iteration (state restoration or update).

5.4.8. Preliminary Observations

Preliminary experimental results demonstrate that EF-TTC [14] offers a substantial advantage in execution time over the original MIP-based solver implemented in Neptune [2]. The MIP approach [21] exhibits exponential complexity, approximately $O(2^{F \cdot N})$,

which becomes infeasible as the system scales, especially for large communities with high numbers of functions and nodes.

EF-TTC, by contrast, guarantees a polynomial upper bound of $O(F^2 \cdot N^3)$. Although still non-trivial, this growth rate is significantly more manageable, allowing the algorithm to scale to realistic workloads.

Practically, the EF-TTC approach performs even better than its worst-case bound thanks to:

- Early pruning of functions with no further gain.
- Dynamic exclusion of infeasible (function, node) pairs.
- Efficient traversal of sparse preference graphs.

The figures below compare the growth trends of the two approaches. The surface that rises sharply represents the exponential complexity of the Neptune MIP-based solver. The flatter surface below corresponds to the polynomial complexity of EF-TTC. The black curve marks the boundary where both methods exhibit approximately equal computational costs. Below this curve, EF-TTC provides better scalability.

Legend:

- **Neptune MIP (Exponential):** represented by the taller surface (colored in orange).
- **EF-TTC (Polynomial):** represented by the flatter surface below (colored in light-blue).
- **Black curve:** shows the threshold where $O(F^2 \cdot N^3) \approx O(2^{F \cdot N})$.

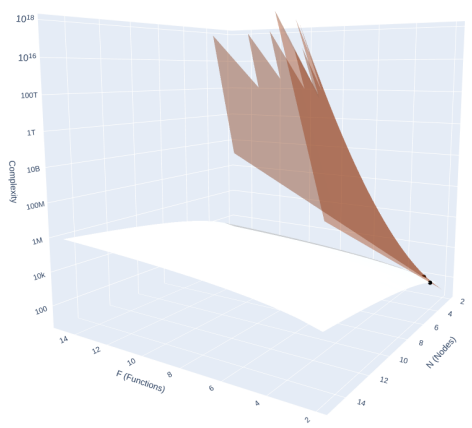

Figure 5.22: EF-TTC vs MIP – angled view

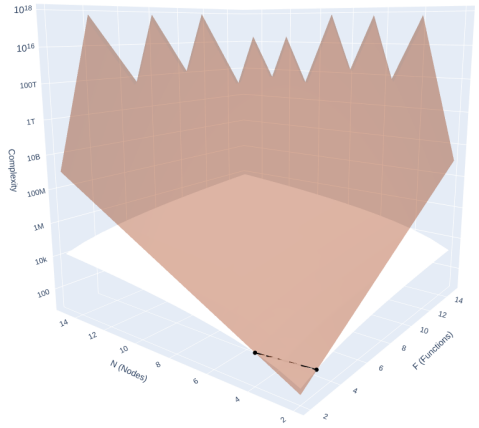

Figure 5.23: EF-TTC vs MIP – front view

5.5. Evaluation Metrics

To evaluate the effectiveness of each proposed method, I performed experiments using metrics tailored to the nature of each algorithm:

- **For clustering algorithms** (both overlapped and resource-aware), the evaluation was based on:
  - *Cluster compactness:* measured through different intra-cluster distance formulas.
  - *Computation time:* empirical execution time over varied inputs.
  - *Visual inspection:* qualitative verification of cluster distributions, assessed via 2D visualizations to check consistency and separation.
- **For the function placement algorithm (EF-TTC)**, the evaluation considered:
  - *Computation time:* total runtime for solving the placement instance.
  - *Objective score:* the final value of the optimization objective (e.g., delay, utilization, or combined).
  - *Constraint satisfaction:* verification that all hard constraints (e.g., memory, capacity) are respected throughout the solution process.

These metrics enabled a comparative study of the proposed solutions in terms of both system performance and scalability.

6 | Results Analysis

This chapter presents and analyzes the results obtained from the experimental evaluation of the proposed algorithms. The discussion is structured based on the nature of the techniques evaluated, distinguishing between clustering strategies and function placement methods. The analysis covers both simulated scenarios and experiments using real or partially realistic datasets, and it is guided by the key evaluation metrics defined in the previous section.

6.1. Clustering Algorithms

6.1.1. Evaluation Metrics

The clustering algorithms are evaluated using three aspects: compactness metrics, computation time, and visual inspection.

Compactness Metrics

Table 6.1 summarizes the metrics used to evaluate the compactness of the resulting clusters. The first three metrics (`mean_pairwise`, `mean_centroid`, `radius`) measure, respectively, the average distance between all node pairs, the average distance from each node to the cluster centroid, and the maximum distance from the centroid. The remaining metrics (`density`, `compactness_pairwise`, `compactness_centroid`, and `compactness_radius`) represent inverse measures of the previous ones or are based on the convex hull area, aiming to assign higher values to more compact clusters. Higher compactness generally indicates better internal cohesion and implies that resources or requests within a cluster can be managed more efficiently and with lower latency.

Metric	Description	Formula (inline)
`mean_pairwise`	Average distance between all node pairs in the cluster.	$\frac{1}{\binom{n}{2}} \sum_{1 \le i < j \le n} d_{ij}$, where d_{ij} is the distance between nodes i and j
`mean_centroid`	Average distance of nodes from the cluster centroid.	$\frac{1}{n} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{c}\|$, with $\mathbf{c} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$
`radius`	Maximum distance of a node from the cluster centroid.	$\max_{1 \le i \le n} \|\mathbf{x}_i - \mathbf{c}\|$, with $\mathbf{c}$ as above
`density`	Cluster density based on convex hull area.	$\frac{n}{\text{Area}(\text{ConvexHull}(\{\mathbf{x}_i\})) + \varepsilon}$
`compactness_pairwise`	Inverse of the mean pairwise distance.	$\frac{1}{\texttt{mean_pairwise} + \varepsilon}$
`compactness_centroid`	Inverse of mean centroid distance.	$\frac{1}{\texttt{mean_centroid} + \varepsilon}$
`compactness_radius`	Inverse of maximum radius.	$\frac{1}{\texttt{radius} + \varepsilon}$

Table 6.1: Compactness metrics for clustering evaluation (higher values indicate better compactness, unless otherwise noted).

6.1.2. Experimental Setup

The experimental evaluation was conducted on two types of network topologies: a fully simulated topology and one derived from real-world data.

Fully simulated topology

In the synthetic scenario, the network was composed of 2,500 nodes randomly placed over a 100×100 grid. This setup emulates the typical spatial heterogeneity found in edge computing infrastructures. The initial topology is shown in Figure 6.1, while Figure 6.2 highlights the same nodes with circle sizes proportional to the resource availability, which was initialized using a uniform distribution.

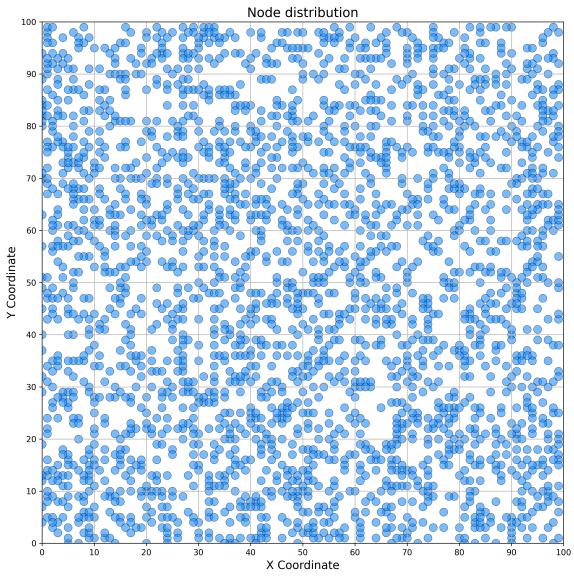

Figure 6.1: Node layout of the fully simulated network.

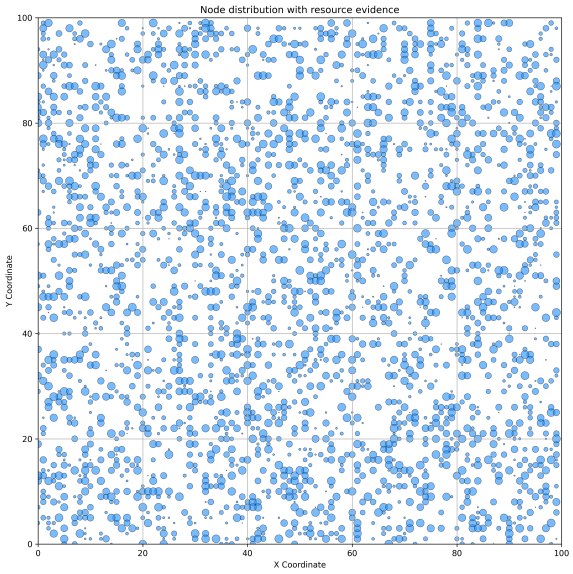

Figure 6.2: Simulated network with resource values: circle size represents node capacity (random distribution).

Real-world topology

For the real-world topology, the T-Drive dataset[1] was utilized. The dataset contains GPS trajectories of over 10,000 taxi drivers in Beijing. The initial coordinates of each taxi were extracted and treated as edge nodes. Figure 6.3 shows the node layout derived from the dataset.

[1] `https://www.kaggle.com/datasets/arashnic/tdriver`

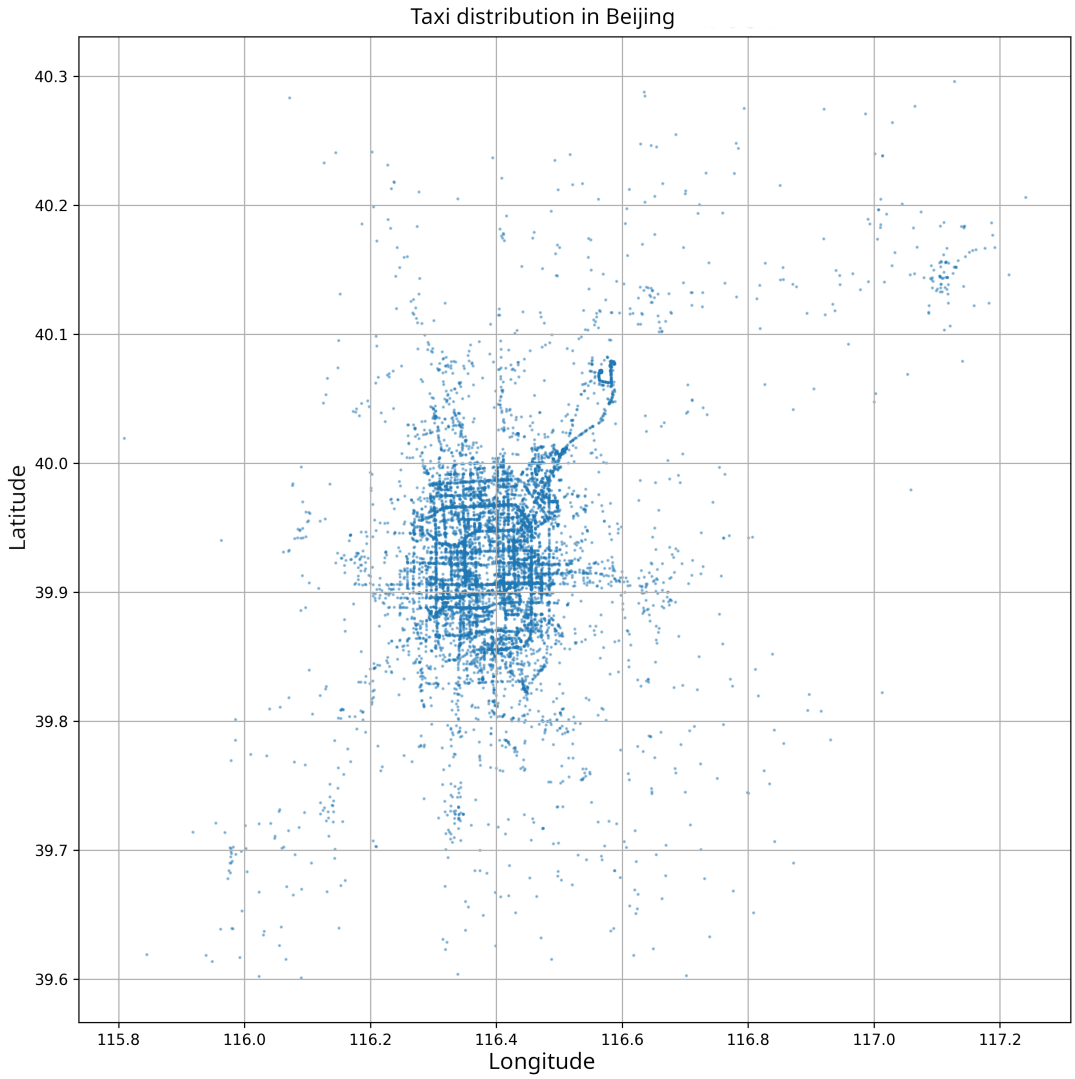

Figure 6.3: Node layout derived from the T-Drive[2] dataset (real-world mobility).

Resource values for the real-world network were assigned according to two different statistical models. The first, a log-normal distribution, simulates the presence of a small number of high-capacity nodes within a majority of low-capacity ones (Figure 6.4). The second, an exponential distribution, models environments dominated by resource-constrained nodes with few exceptions (Figure 6.5). In both plots, circle sizes are proportional to the amount of resources available at each node.

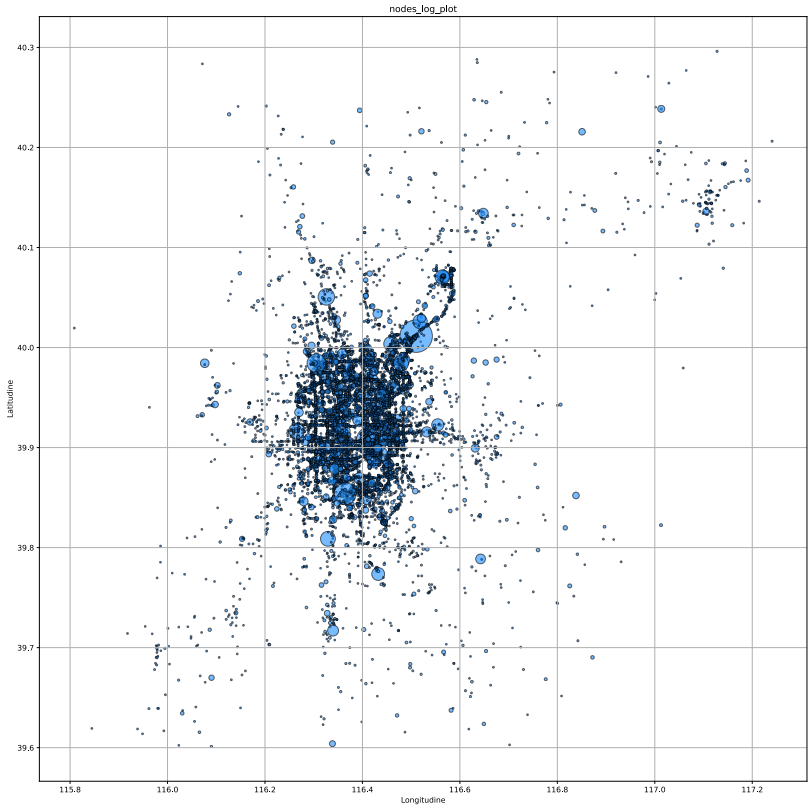

Figure 6.4: Real-world network with log-normal resource distribution: circle size proportional to node capacity.

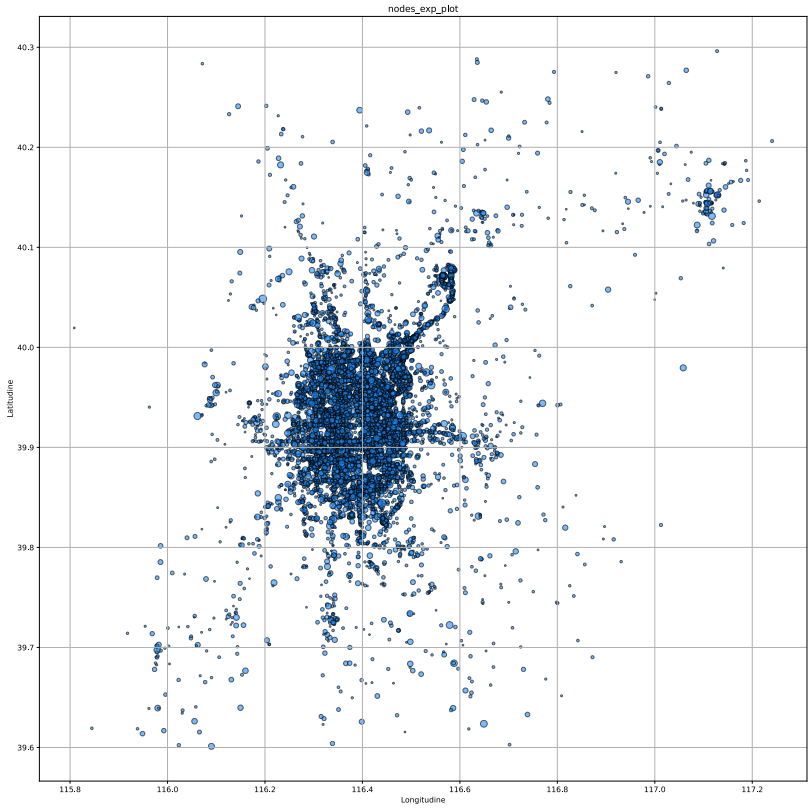

Figure 6.5: Real-world network with exponential resource distribution: circle size proportional to node capacity.

6.1.3. Cluster Compactness

Cluster compactness was evaluated across different algorithms to assess the quality of the partitions generated on both simulated and real-world network topologies. These measures provide insight into the spatial cohesion and structural efficiency of each clustering approach.

Simulated Network Analysis. Figure 6.6 illustrates the comparison of compactness scores on the fully simulated network. As reported in Table 6.2, both the overlapping clustering approach and the resource-aware balanced clustering significantly outperform the baseline SLPA method [18, 22] across all compactness metrics.

The improved compactness observed in both the overlapping and resource-aware clustering strategies primarily stems from the use of a centroid-based assignment approach as in k-means algorithm [1]. This methodology inherently favors the formation of spatially coherent clusters by minimizing the distance between nodes and their respective centroids. In the overlapping strategy, the inclusion of boundary-sharing nodes further enhances this effect by softening cluster borders, resulting in smoother transitions and increased cohesion. Similarly, the resource-aware approach combines centroid proximity with resource distribution constraints, producing clusters that are not only spatially compact but also balanced in terms of computational capacity. This dual consideration leads to consistently strong performance across all compactness metrics, particularly in terms of density and intra-cluster radii.

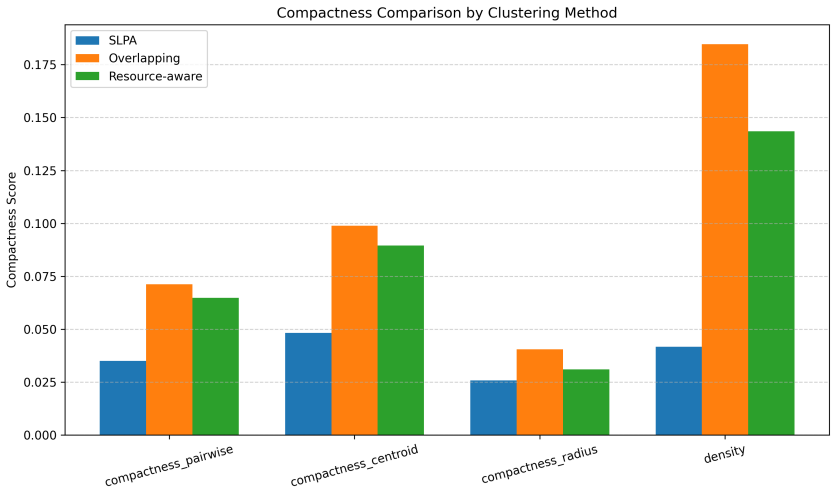

Figure 6.6: Comparison of compactness scores across clustering methods on simulated network data.

Table 6.2: Compactness metrics for clustering on simulated network data.

Metric	SLPA	Overlapping	Resource-aware
compactness_pairwise	0.0351	0.0713	0.0648
compactness_centroid	0.0482	0.0989	0.0895
compactness_radius	0.0258	0.0405	0.0310
density	0.0417	0.1846	0.1435

Real-World Network Analysis. Figure 6.7 and Table 6.3 report the compactness results obtained when applying the clustering algorithms to the real-world node topology extracted from the T-Drive dataset[3]. In this case, the resource-aware approach was evaluated under two different resource assignment models: exponential and log-normal distributions.

While all methods exhibit increased absolute values due to the scale and complexity of the real dataset, the relative performance trend remains consistent. The overlapping clustering again improves compactness compared to SLPA [18, 22], particularly regarding spatial cohesion. The resource-aware method demonstrates superior results across all metrics, with the log-normal variant slightly outperforming the exponential one in terms of compactness centroid and compactness radius, indicating improved internal distribution of resources and spatial grouping.

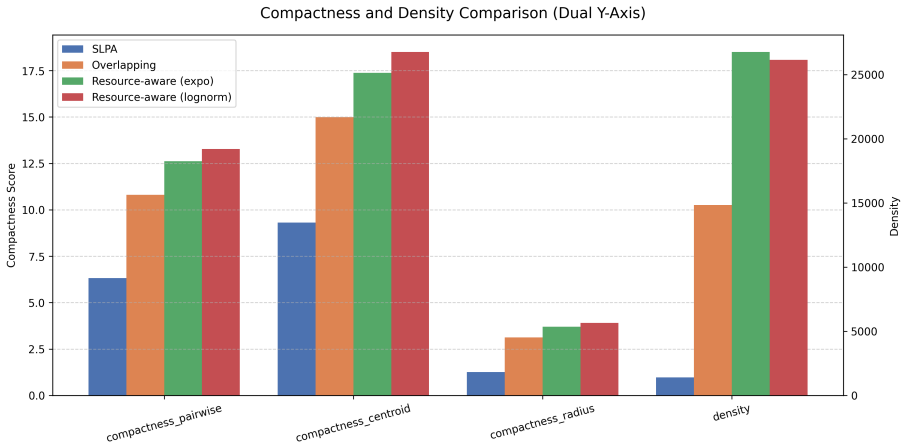

Figure 6.7: Comparison of compactness scores across clustering methods on real-world network data.

[3] https://www.kaggle.com/datasets/arashnic/tdriver

Table 6.3: Compactness metrics for clustering on real-world network data.

Metric	SLPA	Overlapping	Resource-aware (exp)	Resource-aware (lognorm)
compactness_pairwise	6.3224	10.8046	12.6159	13.2741
compactness_centroid	9.3119	14.9864	17.3855	18.5047
compactness_radius	1.2607	3.1314	3.6972	3.9043
density	1408.0963	14839.6579	26757.2238	26152.1679

In conclusion, although both proposed clustering techniques outperform SLPA in terms of compactness metrics, the key achievement lies in the ability to preserve (and even improve) the spatial coherence of clusters while integrating additional structural constraints. Specifically, the overlapping strategy and the resource-aware clustering successfully incorporate shared boundary nodes and resource balancing, respectively, without compromising spatial organization.

6.1.4. Computation Time

Experimental results show that the proposed algorithms exhibit significantly better computational efficiency compared to the SLPA baseline [18, 22]. This improvement is particularly evident in the resource-aware clustering approach, which, despite incorporating a resource balancing mechanism, employs an optimized centroid-based assignment strategy using priority queues. This enables the algorithm to scale efficiently and handle large network topologies with minimal computational overhead.

Simulated Network. Figure 6.8 and Table 6.4 report the computation times for the clustering algorithms on the fully simulated network. In this setting, only the algorithm's core computation time was measured, excluding data initialization and preprocessing phases. Under these conditions, both novel algorithm complete in less time than original SLPA [18, 22]. A more comprehensive breakdown, including initialization overhead, is presented in the analysis of the real-world scenario that follows.

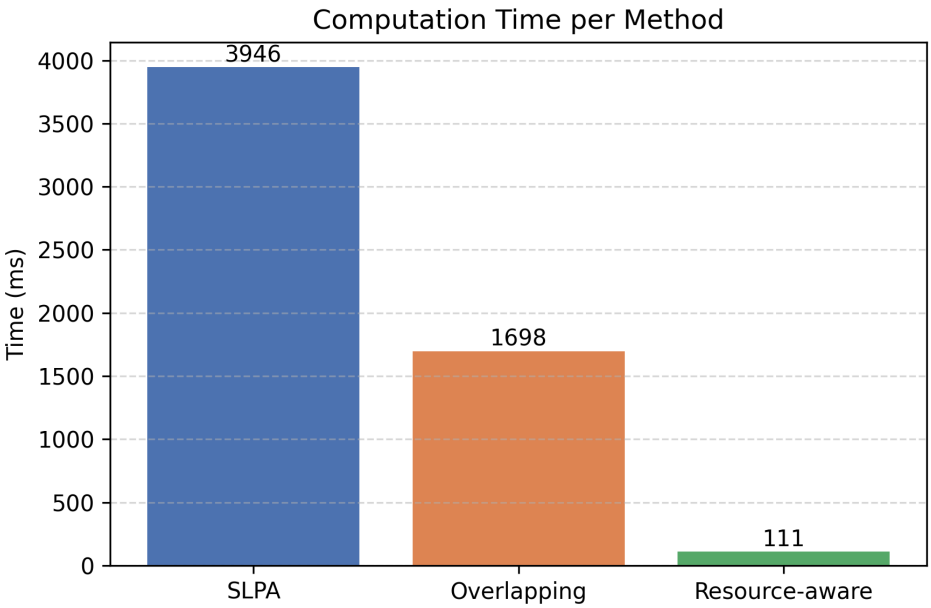

Figure 6.8: Computation time for different clustering algorithms on simulated network data.

Table 6.4: Computation time for clustering algorithms on simulated network data, expressed in milliseconds, seconds, and minutes.

Method	**SLPA**	**Overlapping**	**Resource-aware**
computation_time (ms)	3946	1698	111
computation_time (s)	3.946	1.698	0.111
computation_time (min)	0.0658	0.0283	0.00185

Real-World Network. Figures 6.9 and 6.10, together with Table 6.5, report the execution times measured on the real-world network. In this more comprehensive evaluation, the total time includes both data initialization and clustering execution. SLPA [18, 22] exhibits a total execution time exceeding 3 million milliseconds, largely due to costly preprocessing operations. In contrast, the proposed methods maintain high efficiency even at this scale. The resource-aware clustering approaches (both exponential and log-normal variants) complete the entire pipeline in approximately 133,000 milliseconds, representing an improvement of more than one order of magnitude over the baseline.

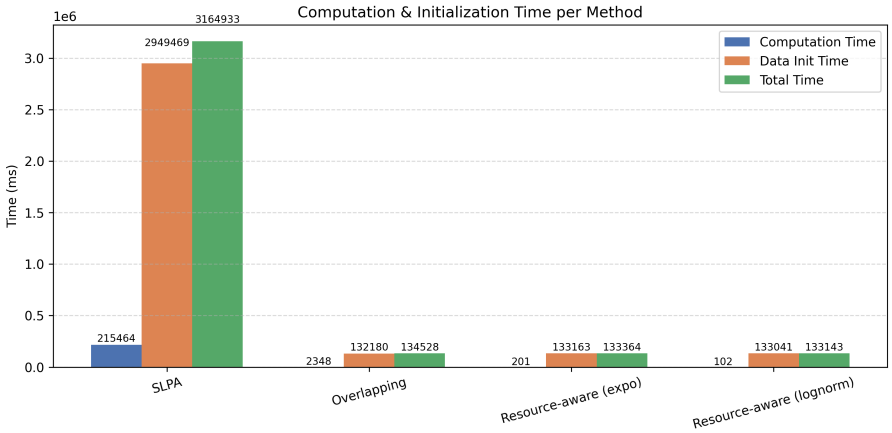

Figure 6.9: Total execution time for clustering algorithms on real-world network data.

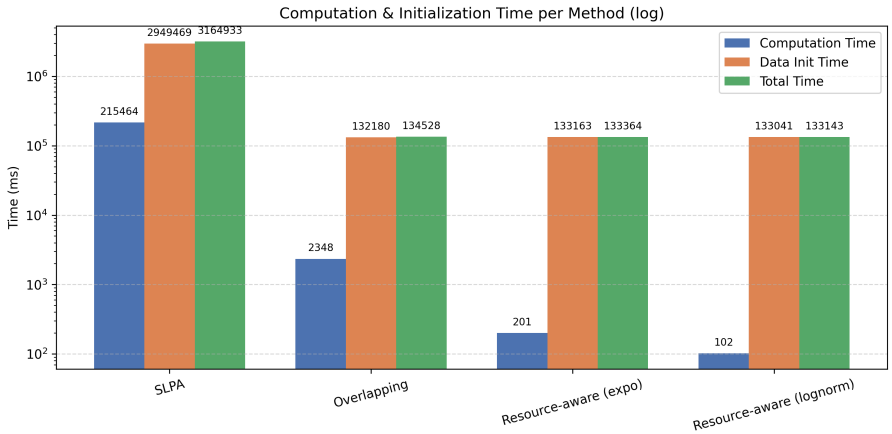

Figure 6.10: Computation time comparison on real-world data (logarithmic scale).

Table 6.5: Execution time breakdown on real-world network data, expressed in milliseconds, seconds, and minutes.

Metric	**SLPA**	**Overlapping**	**Resource-aware (exp)**	**Resource-aware (lognorm)**
computation_time (ms)	215464	2348	201	102
computation_time (s)	215.464	2.348	0.201	0.102
computation_time (min)	3.591	0.0391	0.00335	0.0017
data_initialization_time (ms)	2949469	132180	133163	133041
data_initialization_time (s)	2949.469	132.180	133.163	133.041
data_initialization_time (min)	49.158	2.203	2.219	2.217
total_time (ms)	3164933	134528	133364	133143
total_time (s)	3164.933	134.528	133.364	133.143
total_time (min)	52.750	2.242	2.223	2.219

6.1.5. Visual Inspection

The spatial configuration of the resulting clusters was also assessed through qualitative visual inspection. The following figures present the clustering outputs on both the simulated and real-world network topologies.

Simulated Network. Figures 6.11, 6.12, and 6.13 illustrate the clustering results on the 2,500-node grid. The SLPA baseline (Figure 6.11) shows irregularly shaped and often fragmented communities. In contrast, the proposed overlapping strategy with shared boundary nodes (Figure 6.12) yields more contiguous and compact regions. The resource-aware clustering (Figure 6.13) produces balanced clusters that maintain spatial coherence while accounting for node capacities.

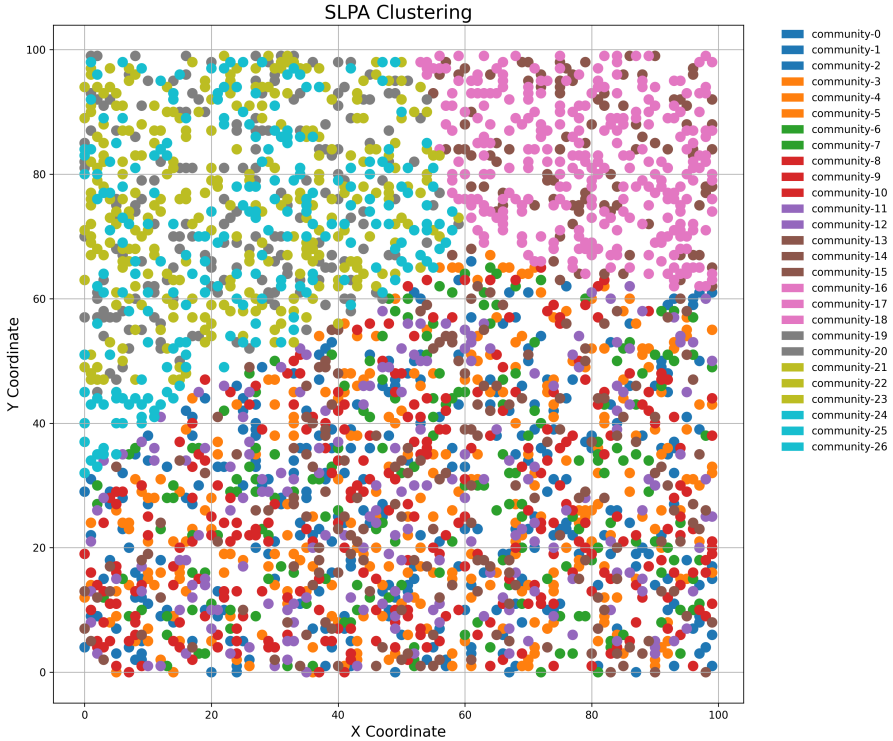

Figure 6.11: SLPA-based overlapping community detection (baseline approach).

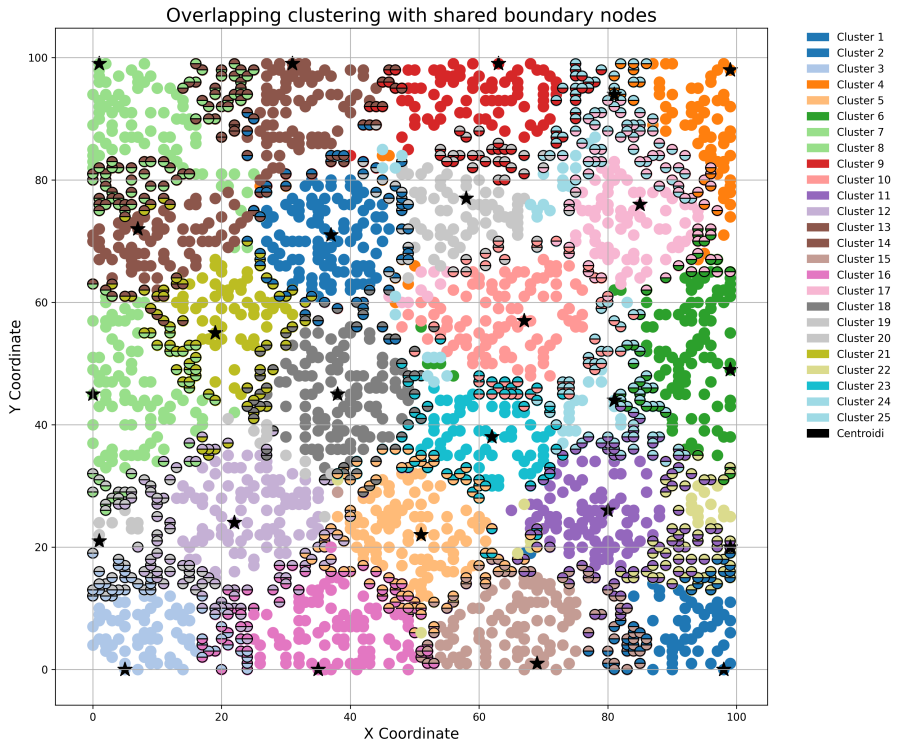

Figure 6.12: Proposed overlapping clustering with shared boundary nodes.

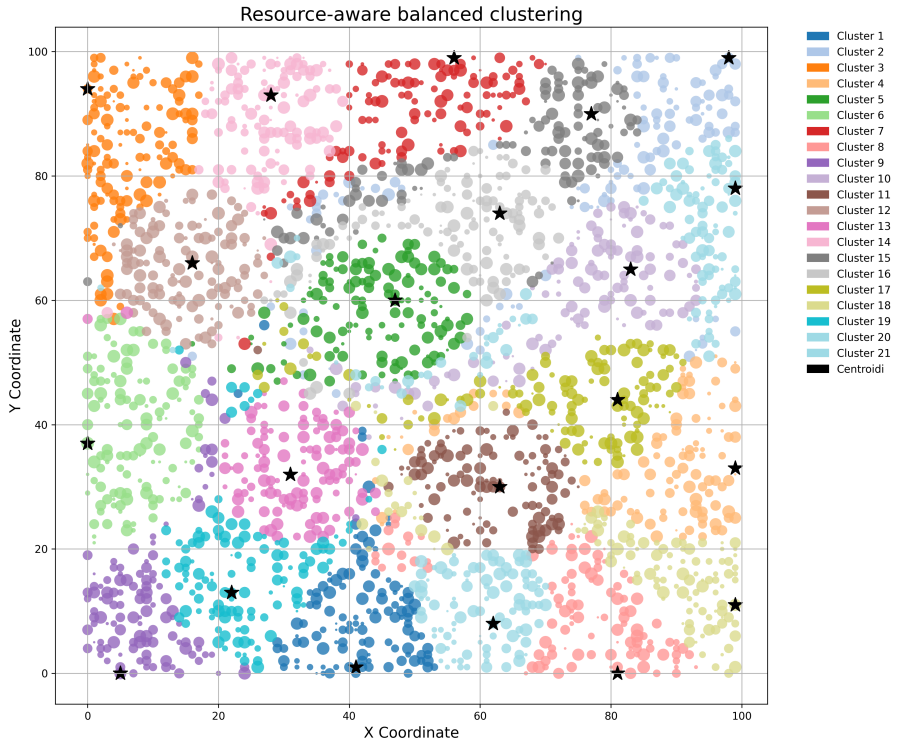

Figure 6.13: Resource-aware balanced clustering strategy: node size represents node resources.

Real-World Network. Figures 6.14 to 6.18 present the clustering outputs obtained from real-world network data derived from taxi mobility traces. The SLPA baseline (Figure 6.14) results in fragmented and irregularly distributed communities, a tendency further illustrated in the close-up view of the clustering structure shown in Figure 6.15. In contrast, the overlapping method (Figure 6.16) demonstrates improved spatial continuity and boundary management, with Figure 6.17 highlighting the precise allocation of shared nodes at community borders. Finally, the resource-aware clustering strategy is shown under two different resource distribution models: exponential (Figure 6.19) and log-normal (Figure 6.18) confirming its ability to maintain spatial coherence while embedding computational balancing.

These visualizations confirm the trends observed in the quantitative analysis. SLPA tends to generate fragmented and heterogeneous clusters, whereas the proposed methods lead to more compact, spatially coherent, and balanced partitions. In addition to improved compactness and execution time, the new strategies successfully implement boundary node sharing and resource balancing, key mechanisms to improve workload distribution and operational efficiency in edge computing communities.

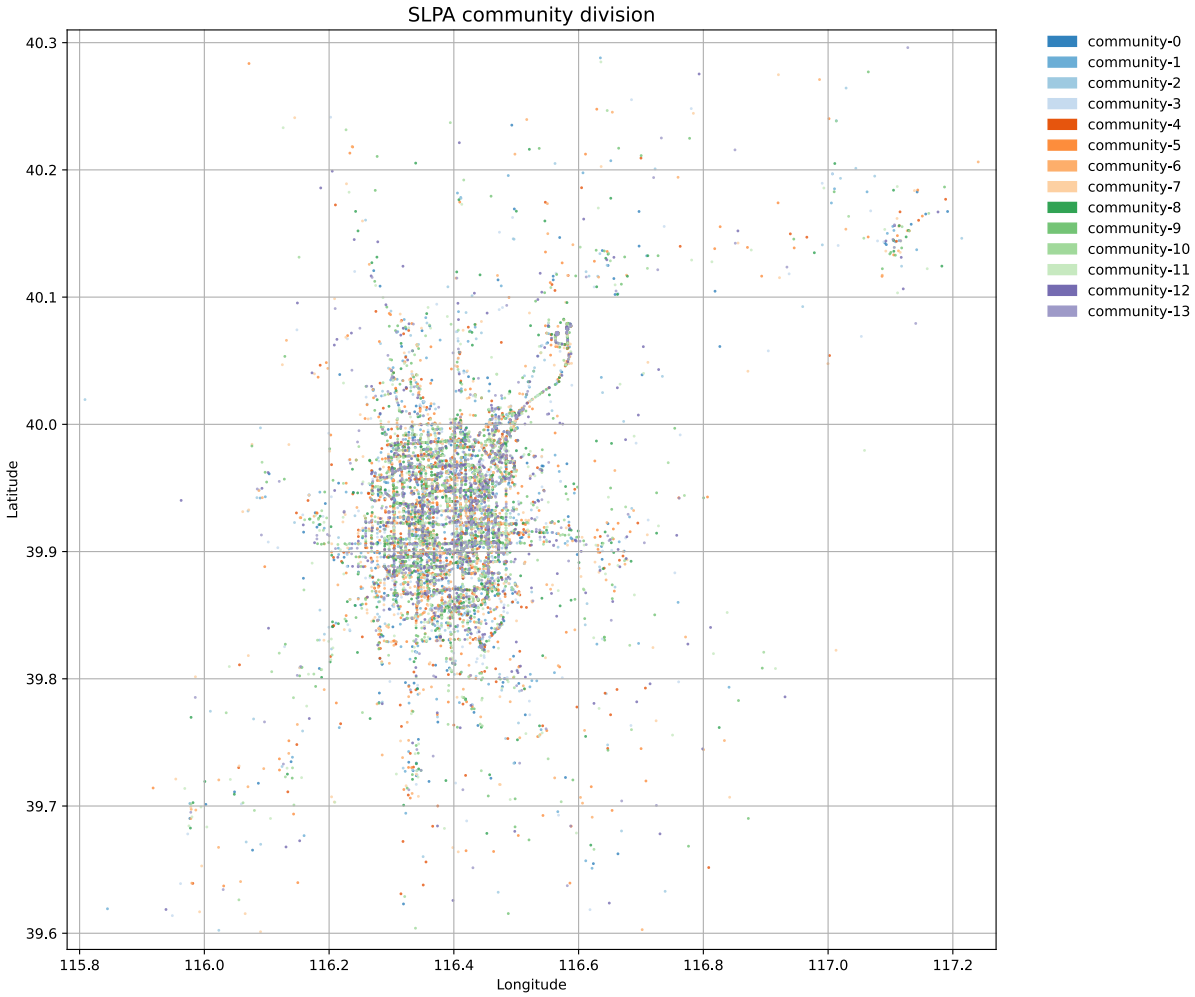

Figure 6.14: SLPA clustering on real-world network data.

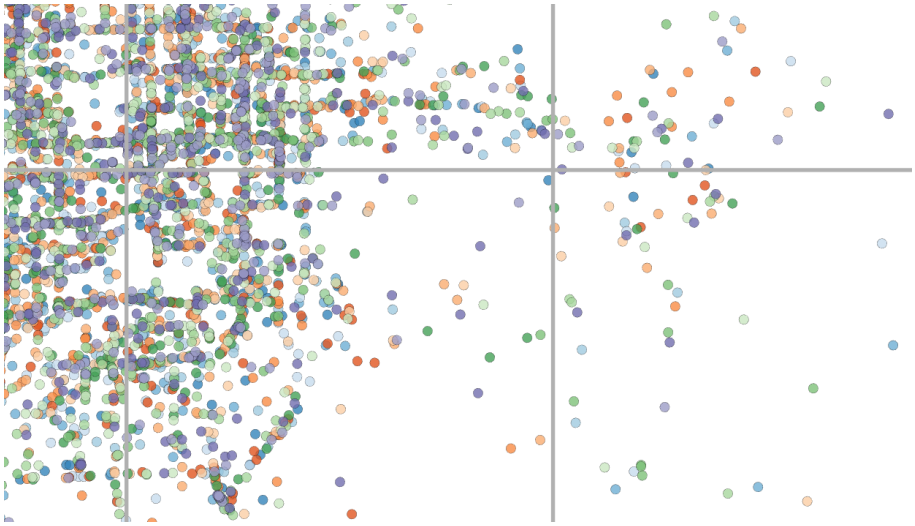

Figure 6.15: Detail of SLPA communities.

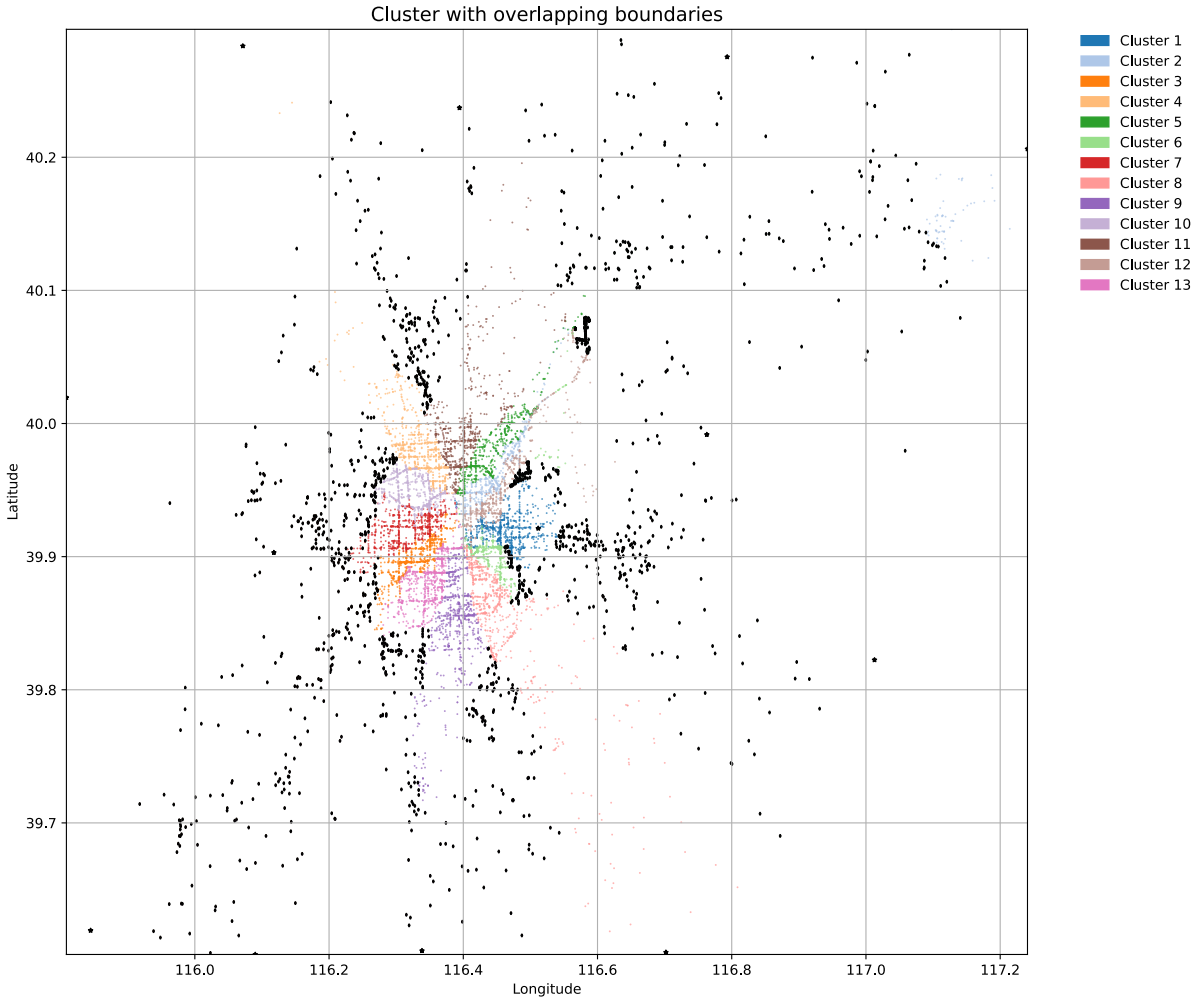

Figure 6.16: Overlapping clustering on real-world data.

Figure 6.17: Detail of boundary node allocation.

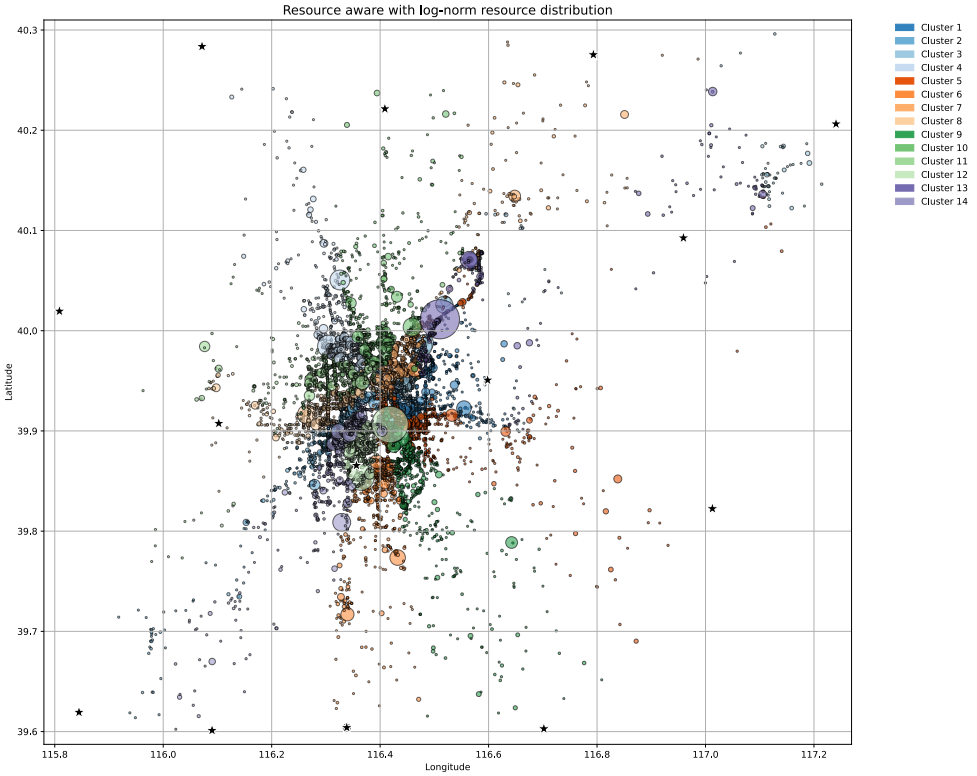

Figure 6.18: Resource-aware clustering (log-normal resource distribution).

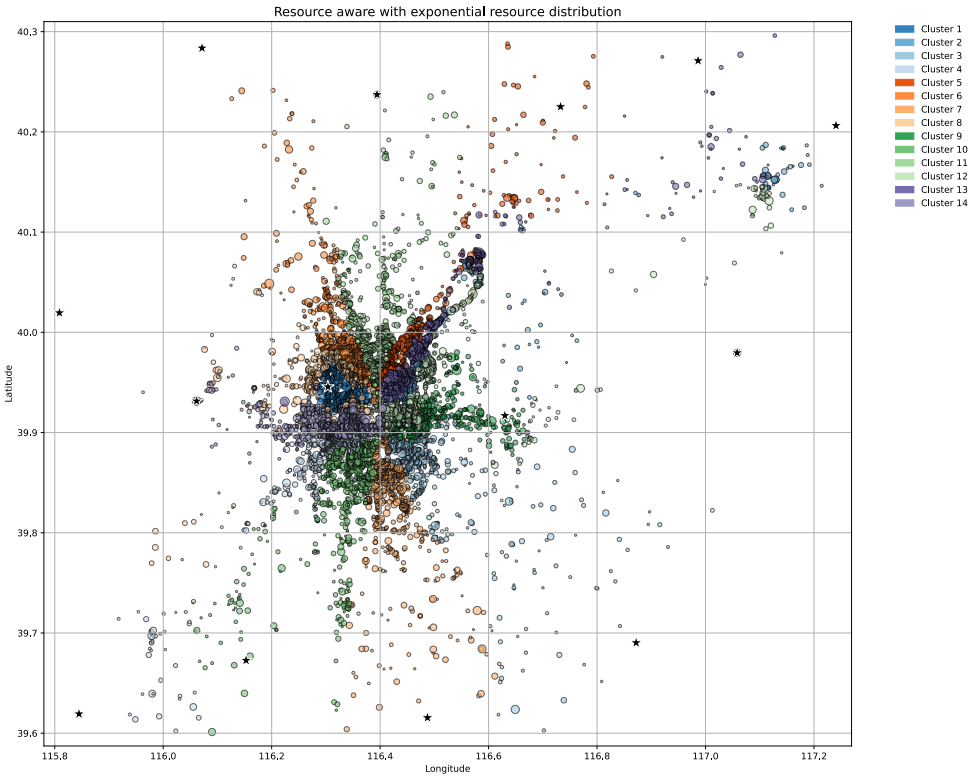

Figure 6.19: Resource-aware clustering (exponential resource distribution).

6.2. Function Placement with EF-TTC

This section presents the evaluation of the EF-TTC algorithm [14] for function placement in edge computing environments. EF-TTC (Edge FaaS-Top Trading Cycles) was tested under two major categories of scenarios: fully simulated test sets and real-world configurations derived from cloud trace data. The evaluation aims to assess the algorithm's efficiency and placement quality with respect to computation time, objective score, and constraint satisfaction.

6.2.1. Evaluation Metrics

The EF-TTC algorithm [14] was evaluated using three key metrics across all optimization configurations (MinDelay, MinUtilization, and MinDelayAndUtilization). Computation time measures algorithmic efficiency. The objective score evaluates either latency reduction, resource efficiency, or their balance (via parameter α). Finally, constraint satisfaction verifies compliance with system limits (CPU, memory, network) for all configurations, demonstrating the algorithm's robustness under different optimization strategies.

6.2.2. Experimental Setup

Fully Simulated Test Sets

A total of 10 simulated test sets were created to evaluate EF-TTC under controlled scalability as shown in table 6.6 . These configurations span a range of increasing complexity:

- Test 0–3: small-scale cases with 1 node and 1–2 functions in various allocation states;
- Test 4–6: 1 or multiple nodes with 2 or more functions, simulating partial and full assignment coverage;
- Test 7–9: full-scale synthetic networks with 25–50 nodes and 5–15 functions.

Table 6.6: Legend of simulated test cases.

Test Case	Scenario
0	1 node, 1 function (not allocated)
1	1 node, 1 function (already allocated)
2	1 node, 2 functions (none allocated)
3	1 node, 2 functions (one allocated)
4	1 node, 2 functions (both allocated)
5	20 nodes, 5 functions (none allocated)
6	20 nodes, 5 functions (all allocated)
7	50 nodes, 15 functions (none allocated)
8	50 nodes, 5 functions (none allocated)
9	25 nodes, 15 functions (none allocated)

These test cases were used to systematically explore EF-TTC behavior across varying degrees of size and allocation difficulty.

Real-World Test Set

The evaluation dataset was constructed using traces from the Alibaba Cluster Trace[4] with the following processing pipeline:

- **Node Selection**:
  - 100 unique physical nodes extracted from `machine_meta.csv`
  - Node specifications: CPU cores and memory capacity recorded
- **Function Extraction**:
  - 25 distinct functions identified from the first 500,000 instances in `batch_instance.csv`
  - Memory requirements mapped from `batch_task.csv`
  - CPU allocations reconstructed from instance-to-node mappings
- **Network Modeling**:
  - Delay matrix estimated synthetically (not present in original traces)
  - Maximum function delays set to 100ms as common edge computing constraint

[4] `https://github.com/alibaba/clusterdata`

The resulting test case maintains the original cluster's resource heterogeneity while introducing necessary network parameters for edge computing simulations. The dataset preserves real-world characteristics including:

- Non-uniform node capabilities (varying CPU/memory configurations)
- Actual function memory footprints from production workloads
- Genuine function-to-node allocation patterns

6.2.3. Computation Time

Experimental results show that the proposed EF-TTC algorithm [14] exhibits significantly better computational efficiency compared to baseline approaches.

Simulated Network.
Figures 6.20, 6.21, and 6.22 along with Tables 6.7, 6.8, and 6.9 report computation times across various representative synthetic configurations. Only the logarithmic-scale plots are shown to improve readability and to better visualize differences in small and large values simultaneously.

Under the **MinDelay** objective (Figure 6.20, Table 6.7), EF-TTC consistently outperforms Neptune [2] across all test scenarios, with improvements becoming especially prominent as the number of nodes and functions increases. For instance, in the 50 nodes and 15 functions case, EF-TTC completes in 8.5 seconds, compared to over 76 seconds required by Neptune.

For the **MinUtilization** strategy (Figure 6.21, Table 6.8), EF-TTC maintains its performance advantage. Even in larger deployments such as the 25-node, 15-function case, EF-TTC remains well under 1 second, whereas Neptune [2] takes more than 2 seconds. These improvements scale well with increasing complexity.

Finally, in the **MinDelayAndUtilization** scenario (Figure 6.22, Table 6.9), EF-TTC again delivers highly efficient runtimes. Notably, in the most complex configuration (50 nodes, 15 functions), EF-TTC completes placement in just over 10 seconds, while Neptune [2] takes nearly 95 seconds. This highlights EF-TTC's robustness in multi-objective environments.

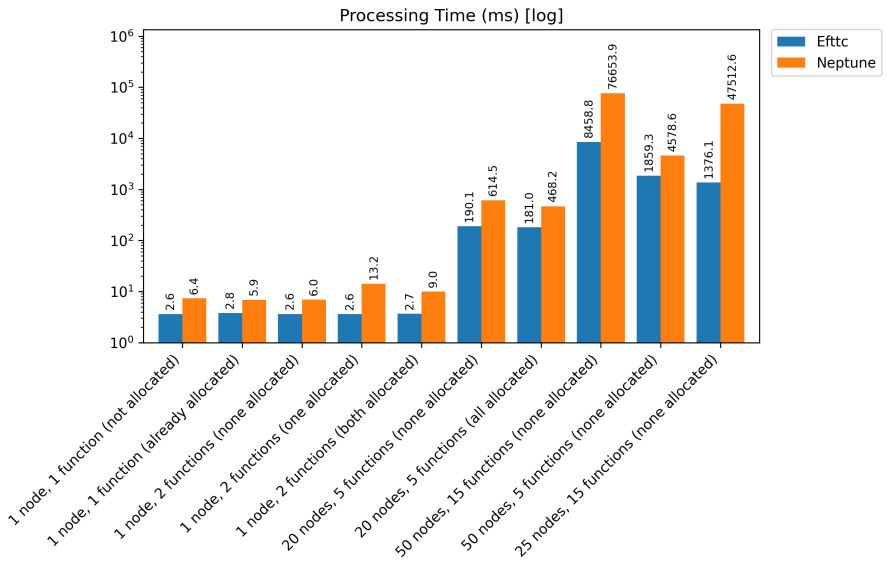

Figure 6.20: Computation time (ms) for MinDelay across synthetic scenarios (log scale).

Table 6.7: Computation time (ms) for all synthetic test cases (MinDelay).

Scenario	**EF-TTC**	**Neptune**
1 node, 1 function (not allocated)	2.64	6.35
1 node, 1 function (already allocated)	2.81	5.90
1 node, 2 functions (none allocated)	2.63	5.96
1 node, 2 functions (one allocated)	2.61	13.23
1 node, 2 functions (both allocated)	2.66	8.98
20 nodes, 5 functions (none allocated)	190.08	614.45
20 nodes, 5 functions (all allocated)	181.03	468.19
50 nodes, 15 functions (none allocated)	8458.77	76653.91
50 nodes, 5 functions (none allocated)	1859.27	4578.64
25 nodes, 15 functions (none allocated)	1376.12	47512.61

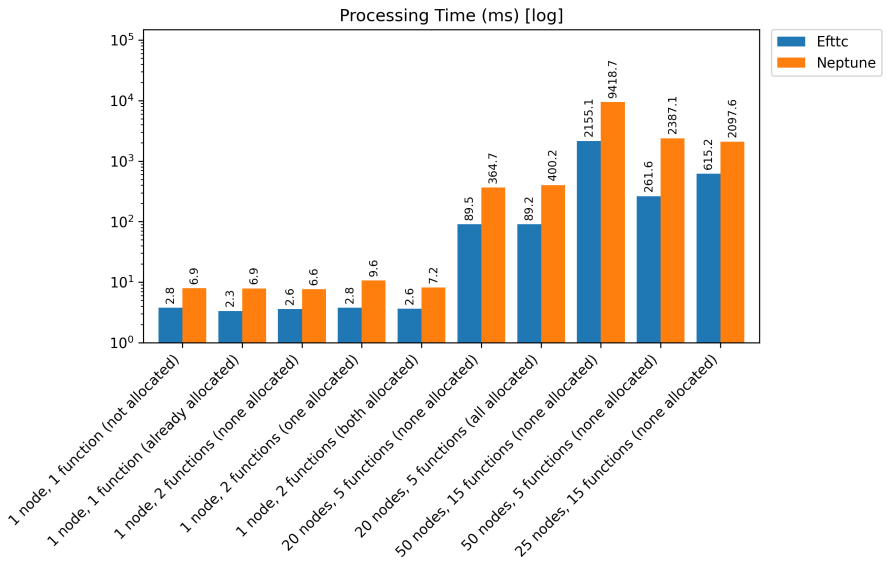

Figure 6.21: Computation time (ms) for MinUtilization across synthetic scenarios (log scale).

Table 6.8: Computation time (ms) for all synthetic test cases (MinUtilization).

Scenario	**EF-TTC**	**Neptune**
1 node, 1 function (not allocated)	2.77	6.91
1 node, 1 function (already allocated)	2.33	6.89
1 node, 2 functions (none allocated)	2.57	6.65
1 node, 2 functions (one allocated)	2.78	9.61
1 node, 2 functions (both allocated)	2.63	7.16
20 nodes, 5 functions (none allocated)	89.54	364.67
20 nodes, 5 functions (all allocated)	89.17	400.24
50 nodes, 15 functions (none allocated)	2155.11	9418.75
50 nodes, 5 functions (none allocated)	261.58	2387.07
25 nodes, 15 functions (none allocated)	615.18	2097.56

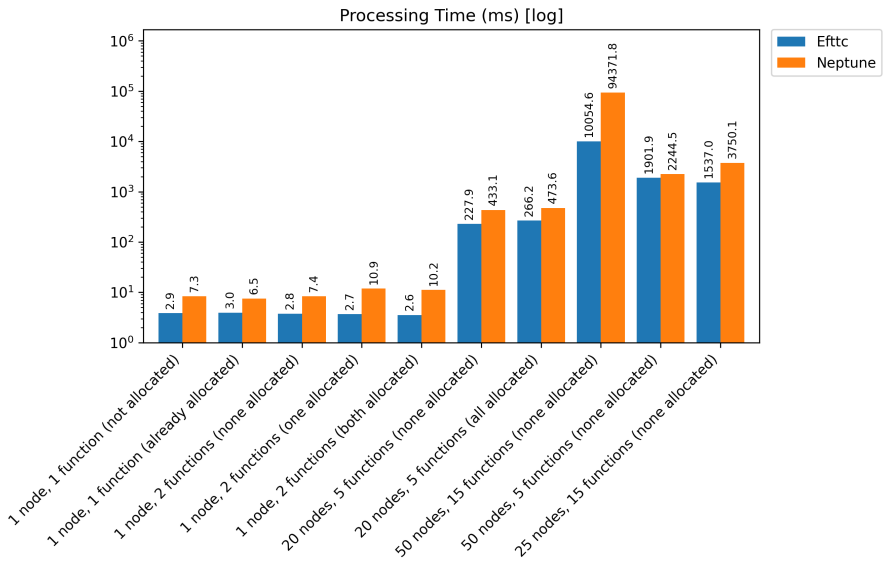

Figure 6.22: Computation time (ms) for MinDelayAndUtilization across synthetic scenarios (log scale).

Table 6.9: Computation time (ms) for all synthetic test cases (MinDelayAndUtilization).

Scenario	**EF-TTC**	**Neptune**
1 node, 1 function (not allocated)	2.86	7.31
1 node, 1 function (already allocated)	2.96	6.54
1 node, 2 functions (none allocated)	2.76	7.36
1 node, 2 functions (one allocated)	2.69	10.87
1 node, 2 functions (both allocated)	2.55	10.20
20 nodes, 5 functions (none allocated)	227.85	433.15
20 nodes, 5 functions (all allocated)	266.12	473.61
50 nodes, 15 functions (none allocated)	10054.62	94371.76
50 nodes, 5 functions (none allocated)	1901.85	2244.49
25 nodes, 15 functions (none allocated)	1536.99	3750.09

Real-World Network.

Figures 6.23, 6.24, and 6.25 and Table 6.10 confirm that EF-TTC maintains its computational efficiency even when facing real-world topologies and realistic function placement scenarios.

The plots show a dramatic reduction in runtime: for example, under the **MinDelayAndUtilization** objective, Neptune's Mixed Integer Programming solver takes over **20 min-**

utes (1.25 million ms), while EF-TTC completes in less than **15 seconds**. This improvement is consistent across all objectives.

Such performance highlights the practical advantage of EF-TTC [14]: it is not only effective but also computationally lightweight. This makes it a viable replacement for solver-based strategies like Neptune [2], especially in environments where scalability is essential. It enables experimentation with larger input sizes, including networks with more nodes and heavier function workloads.

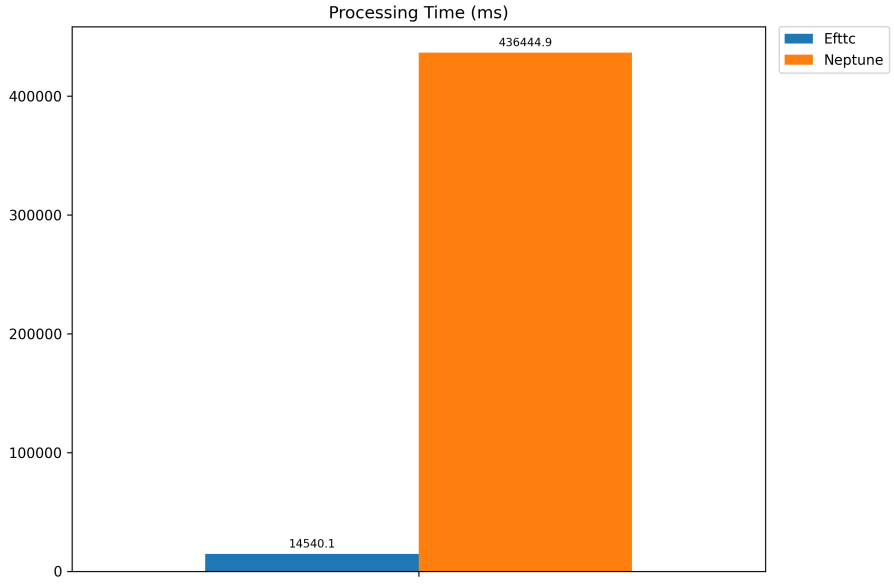

Figure 6.23: Computation time (ms) for MinDelay on real-world dataset.

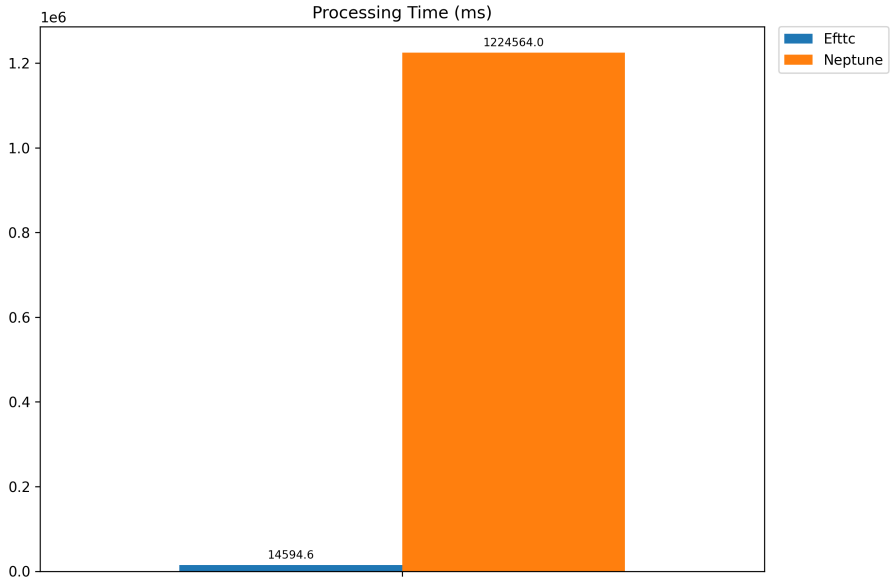

Figure 6.24: Computation time (ms) for MinUtilization on real-world dataset.

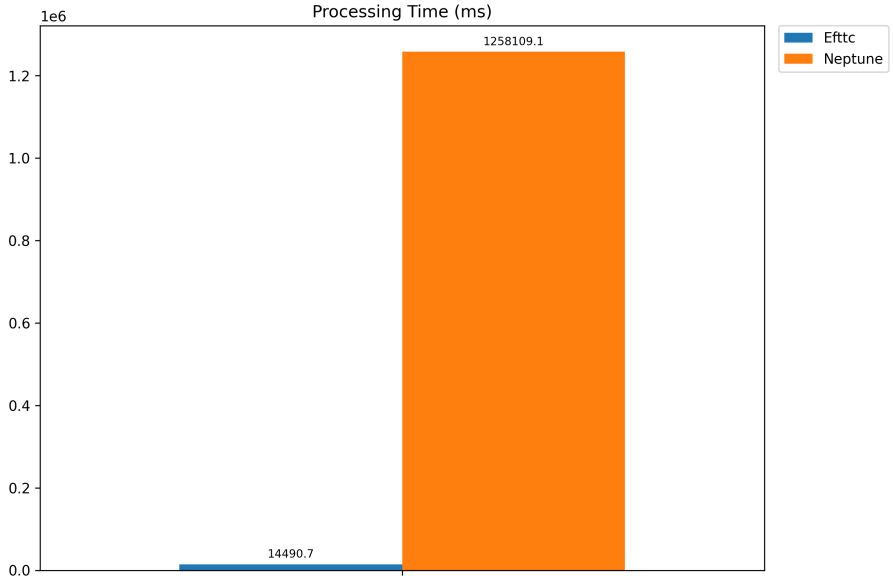

Figure 6.25: Computation time (ms) for MinDelayAndUtilization on real-world dataset.

Table 6.10: Computation time (ms, seconds, and minutes) under each optimization strategy (real-world dataset)

Method	MinDelay	MinUtilization	MinDelayAndUtilization
EF-TTC	14540.1	14594.6	14490.7
Neptune	436444.9	1224564.0	1258109.1
EF-TTC (s)	14.54	14.59	14.49
Neptune (s)	436.44	1224.56	1258.11
EF-TTC (min)	0.24	0.24	0.24
Neptune (min)	7.27	20.41	20.97

6.2.4. Objective Score

EF-TTC [14] effectively optimizes the target placement objective in all evaluated scenarios. The objective function varies depending on the selected optimization goal:

- **MinDelay:** minimizes the weighted sum of end-to-end delays across all function requests;
- **MinUtilization:** minimizes the number of active nodes involved in the deployment;
- **MinDelayAndUtilization:** balances both metrics using a weighted combination controlled by a tunable parameter α.

Note: for all objectives, lower scores indicate better performance. EF-TTC was designed to achieve optimal or near-optimal placements with significantly reduced execution times, as demonstrated in the earlier runtime analysis section.

Across these different goals, EF-TTC consistently achieves competitive or optimal objective scores while ensuring high efficiency and scalability.

Simulated Network.
Tables 6.11, 6.12, and 6.13 present the scores obtained in key synthetic scenarios for each optimization goal. Specifically:

- In the *MinDelay* (Table 6.11) and *MinUtilization* (Table 6.12) configurations, EF-TTC and Neptune achieve identical scores across all scenarios, confirming EF-TTC's ability to match the optimal solutions found by Neptune.
- Under *MinDelayAndUtilization* (Table 6.13), EF-TTC yields slightly worse scores only in the scenarios with 15 functions (0.27 vs 0.05 and 0.28 vs 0.1), yet still remains

close to optimal. This indicates a minor trade-off in accuracy for a significant gain in efficiency.

Table 6.11: Objective score across selected synthetic scenarios (MinDelay).

Scenario	**EF-TTC**	**Neptune**
1 node, 1 function (not allocated)	0.0	0.0
1 node, 1 function (already allocated)	0.0	0.0
1 node, 2 functions (none allocated)	0.0	0.0
1 node, 2 functions (one allocated)	0.0	0.0
1 node, 2 functions (both allocated)	0.0	0.0
20 nodes, 5 functions (none allocated)	0.0	0.0
20 nodes, 5 functions (all allocated)	0.0	0.0
50 nodes, 15 functions (none allocated)	0.0	0.0
50 nodes, 5 functions (none allocated)	0.0	0.0
25 nodes, 15 functions (none allocated)	0.0	0.0

Table 6.12: Objective score across selected synthetic scenarios (MinUtilization).

Scenario	**EF-TTC**	**Neptune**
1 node, 1 function (not allocated)	1	1
1 node, 1 function (already allocated)	1	1
1 node, 2 functions (none allocated)	1	1
1 node, 2 functions (one allocated)	1	1
1 node, 2 functions (both allocated)	1	1
20 nodes, 5 functions (none allocated)	2	2
20 nodes, 5 functions (all allocated)	1	1
50 nodes, 15 functions (none allocated)	5	5
50 nodes, 5 functions (none allocated)	2	2
25 nodes, 15 functions (none allocated)	5	5

Table 6.13: Objective score across selected synthetic scenarios (MinDelayAndUtilization).

Scenario	EF-TTC	Neptune
1 node, 1 function (not allocated)	0.5	0.5
1 node, 1 function (already allocated)	0.5	0.5
1 node, 2 functions (none allocated)	0.5	0.5
1 node, 2 functions (one allocated)	0.5	0.5
1 node, 2 functions (both allocated)	0.5	0.5
20 nodes, 5 functions (none allocated)	0.25	0.25
20 nodes, 5 functions (all allocated)	0.25	0.25
50 nodes, 15 functions (none allocated)	0.27	0.05
50 nodes, 5 functions (none allocated)	0.25	0.25
25 nodes, 15 functions (none allocated)	0.28	0.1

Real-World Network.
Table 6.14 reports the objective scores obtained on a realistic testset with actual network delays and workload distributions. Results confirm that EF-TTC [14] matches the placement quality achieved by the solver-based Neptune method while providing drastically faster runtimes (see Table 6.10).

Table 6.14: Objective score under each optimization strategy (real-world dataset)

Method	MinDelay	MinUtilization	MinDelayAndUtilization
EF-TTC	0.0	1	0.0
Neptune	0.0	1	0.0

6.2.5. Constraint Satisfaction

EF-TTC [14] satisfies all imposed constraints during placement computation. These constraints include:

- **CPU limit:** ensures that the total core usage on each node does not exceed the available cores;
- **Memory limit:** ensures that the total memory usage by deployed functions on a node does not exceed its memory capacity;
- **Request handling:** ensures that every request is properly handled or rerouted according to node availability and prior assignments;

- **Budget respect:** ensures that the total cost of activated nodes remains within the specified budget;
- **c_x consistency:** ensures consistency between the function deployment variable $c[f, j]$ and the routing variable $x[i, f, j]$;
- **n_c consistency:** ensures consistency between the node activation variable $n[i]$ and function deployments $c[f, i]$.

Simulated Network.
All tests on synthetic datasets confirmed full constraint satisfaction. In every configuration and scenario tested, EF-TTC [14] adhered strictly to all defined constraints without exception. This comprehensive compliance was programmatically validated during execution, ensuring feasibility and correctness of the placements.

Real-World Network.
Also in real-world test cases, EF-TTC [14] consistently met every constraint category. From CPU and memory limits to request redirection logic and budget constraints, every individual requirement was validated and satisfied. Notably, the structural consistency constraints C_x and n_c were also respected across all placements, confirming EF-TTC's robustness and reliability in realistic scenarios.

Summary Table.
To visually reinforce this result, Table 6.15 reports constraint satisfaction for both methods:

Table 6.15: Constraint satisfaction overview

Constraint	Neptune	EF-TTC
c_x consistency	✓	✓
Memory limit	✓	✓
Request handling	✓	✓
CPU limit	✓	✓
n_c consistency	✓	✓
Budget respect	✓	✓

No violations were detected in any configuration, further confirming the algorithm's correctness and practical viability.

6.3. Summary of Results

The experimental evaluation confirms the validity and efficiency of the proposed methods in overcoming resource saturation and optimizing function placement within edge computing communities.

- **Clustering Strategies:** The newly designed clustering algorithms significantly improve over the original Neptune version [2] in both quality and performance. They generate more compact communities in less time, thanks to two key innovations:
  - The *Shared Nodes Clustering Algorithm* introduces boundary-overlapping clusters, allowing nodes to belong to multiple communities. This enhances resilience, improves resource availability at the edges, and enables flexible rerouting under load.
  - The *Balanced Resource-Aware Clustering Algorithm* departs from traditional node-count balancing, instead distributing nodes based on actual available resources, resulting in more stable and scalable communities under diverse workloads.
- **EF-TTC Function Placement:** The EF-TTC location algorithm [14] consistently satisfies all imposed restrictions, including CPU and memory limits, request handling, budget respect and variable consistency (c_x, n_c) in all scenarios. Although the resulting placements are not always perfectly optimal, they are near-optimal and produced in dramatically reduced time compared to solver-based approaches. In complex real-world settings, EF-TTC completes execution in seconds rather than minutes, demonstrating speedups of one order of magnitude.

These results demonstrate that the proposed methods not only improve theoretical performance metrics but also translate into tangible improvements in real-world deployments.

7 | Conclusions and future developments

This thesis has addressed the critical issue of resource/community saturation in edge computing communities managed by the NEPTUNE framework [2]. To this end, we explored, designed, and implemented several alternative solutions, each targeting a different aspect of the problem. The results obtained from the analysis and testing of these strategies demonstrated their effectiveness and potential applicability in real-world MEC scenarios.

All proposed algorithms were developed under the assumption that the other components of NEPTUNE operate as black boxes. This assumption was intentional: by treating inputs as fixed and outputs as structured according to the expected NEPTUNE interface, the new modules could be developed in isolation while ensuring seamless integration. This design philosophy facilitates component interchangeability and simplifies system upgrades, promoting modularity and maintainability.

The three main strategies investigated were:

1. **Overlapping Clustering with Shared Border Nodes:** This approach allows certain nodes to be shared among multiple communities, thus enhancing load balancing across community boundaries and offering greater flexibility in handling transient overload conditions. The algorithm enforces constraints on the number of shared and exclusive nodes per cluster to guarantee resource availability and community independence.

2. **Resource-aware Balanced Clustering:** Unlike conventional clustering based solely on node count, this strategy incorporates node-level resource capacities (e.g., CPU, memory) into the clustering process. By distributing the workload according to actual resource availability, this method ensures fairer and more sustainable load distribution, especially in heterogeneous infrastructures.

3. **EF-TTC (Edge FaaS-Top Trading Cycles) [14]:** This advanced solution focuses on improving the placement of serverless functions by considering business

logic cycles and node topologies. By aligning placement decisions with the natural execution patterns of applications, EF-TTC improves drammatically the execution time allowing the construction of bigger cluster of nodes and reducing the occurence of saturation.

These approaches can be used independently or synergistically, depending on the specific context and system constraints. Combining them could yield even more robust and adaptive solutions to load management at the edge.

Moreover, the modularity and interoperability of the developed code open the path for its direct adoption into the NEPTUNE pipeline. By adhering to NEPTUNE's input/output interface conventions [2], the algorithms can replace existing modules with minimal integration effort. This integration could lead to significant improvements in execution performance, reduction in response times, and most notably, a lower occurrence of saturation-induced failures.

Future Developments

Several directions remain open for future exploration. During the initial analysis phase, other promising approaches were identified but not deeply investigated within this thesis due to time constraints. These include:

- **Predictive Models for Load Management discussed in Section 3.5:** Leveraging machine learning models to forecast demand patterns can enable proactive scaling and resource provisioning. By anticipating traffic spikes, NEPTUNE could initiate mitigation strategies before performance degradation occurs.
- **Advanced Function Offloading discussed in Section 3.3:** While NEPTUNE [2] currently supports inter-node routing within a community, an advanced strategy for offloading functions to other communities represents a logical evolution. This mechanism would introduce an additional level of redundancy and flexibility, especially in cases of persistent overload within a single community.

These enhancements, discussed in Chapter 3, represent natural extensions of this research and are recommended as promising topics for continued development and future thesis work.

Implementation within the NEPTUNE system. Beyond theoretical design and simulation, a crucial future milestone is the concrete integration of the proposed methods into the actual NEPTUNE infrastructure. This would involve aligning the developed

algorithms with NEPTUNE's modular architecture, adapting the routing and scheduling pipeline, and performing validation under real-world constraints such as live traffic, cold starts, and system latency. Such integration would bridge the gap between experimental research and production-ready solutions, enabling practical impact in real MEC deployments.

Bibliography

[1] M. Ahmed, R. Seraj, and S. M. S. Islam. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8), 2020. ISSN 2079-9292. doi: 10.3390/electronics9081295. URL https://www.mdpi.com/2079-9292/9/8/1295.

[2] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano. Neptune: A comprehensive framework for managing serverless functions at the edge. *ACM Transactions on Autonomous and Adaptive Systems*, 19(1):1–32, 2024. ISSN 1556-4665. doi: 10.1145/3634750. URL https://hdl.handle.net/11311/1261497. WOS:001193663100007; Scopus: 2-s2.0-85186542022.

[3] K. A. Benatti, L. G. Pedroso, and A. A. Ribeiro. Fuzzy clustering with capacity constraints: Algorithm, convergence analysis and numerical experiments. *Expert Systems with Applications*, 258:125191, 2024. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2024.125191. URL https://www.sciencedirect.com/science/article/pii/S095741742402058X.

[4] W.-C. Chu and K.-F. Ssu. Location-free boundary detection in mobile wireless sensor networks with a distributed approach. *Computer Networks*, 70:96–112, 2014. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2014.05.005. URL https://www.sciencedirect.com/science/article/pii/S1389128614001959.

[5] C. Cicconetti, M. Conti, and A. Passarella. In-network computing with function as a service at the edge. *Computer*, 55(9):65–73, Sept. 2022. ISSN 1558-0814. doi: 10.1109/mc.2021.3130659. URL http://dx.doi.org/10.1109/MC.2021.3130659.

[6] R. A. da Silva and F. de A.T. de Carvalho. Weighted clusterwise linear regression based on adaptive quadratic form distance. *Expert Systems with Applications*, 185:115609, 2021. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2021.115609. URL https://www.sciencedirect.com/science/article/pii/S095741742101006X.

[7] G. Destino and G. T. F. de Abreu. Network boundary recognition via graph-theory.

In *2008 5th Workshop on Positioning, Navigation and Communication*, pages 271–275, 2008. doi: 10.1109/WPNC.2008.4510385.

[8] P. Franti, O. Virmajoki, and V. Hautamaki. Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1875–1881, 2006. doi: 10.1109/TPAMI.2006.227.

[9] M. Ghorbian and M. Ghobaei-Arani. Function offloading approaches in serverless computing: A survey. *Computers and Electrical Engineering*, 120:109832, 2024. ISSN 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2024.109832. URL `https://www.sciencedirect.com/science/article/pii/S0045790624007596`.

[10] M. Ghorbian, M. Ghobaei-Arani, and R. Asadolahpour-Karimi. Function placement approaches in serverless computing: A survey. *Journal of Systems Architecture*, 157:103291, 2024. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2024.103291. URL `https://www.sciencedirect.com/science/article/pii/S1383762124002285`.

[11] K. Hongchang, W. Hui, S. Hongbin, and H. Yang. Deep reinforcement learning-based task offloading and service migrating policies in service caching-assisted mobile edge computing. *China Communications*, 21(4):88–103, 2024. doi: 10.23919/JCC.fa.2023-0474.202404.

[12] R. Liu, S. Feng, R. Shi, and W. Guo. Weighted graph clustering for community detection of large social networks. *Procedia Computer Science*, 31:85–94, 2014. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2014.05.248. URL `https://www.sciencedirect.com/science/article/pii/S1877050914004256`. 2nd International Conference on Information Technology and Quantitative Management, ITQM 2014.

[13] M. I. Malinen and P. Fränti. Balanced k-means for clustering. In P. Fränti, G. Brown, M. Loog, F. Escolano, and M. Pelillo, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 32–41, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44415-3.

[14] S. P. Moakhar and S. Abrishami. An efficient mechanism for function scheduling and placement in function as a service edge environment. *Journal of Network and Computer Applications*, 226:103890, 2024. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2024.103890. URL `https://www.sciencedirect.com/science/article/pii/S1084804524000675`.

[15] C. Nguyen, C. Klein, and E. Elmroth. Multivariate lstm-based location-aware work-

load prediction for edge data centers. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 341–350, 2019. doi: 10.1109/CCGRID.2019.00048.

[16] E. Oikonomou, S. Plastras, D. Tsoumatidis, D. N. Skoutas, and A. Rouskas. Workload prediction for efficient node management in mobile edge computing. In *2024 IFIP Networking Conference (IFIP Networking)*, pages 461–467, 2024. doi: 10.23919/IFIPNetworking62109.2024.10619775.

[17] J. Palowitch, S. Bhamidi, and A. B. Nobel. Significance-based community detection in weighted networks. *Journal of Machine Learning Research*, 18(188):1–48, 2018. URL `http://jmlr.org/papers/v18/17-377.html`.

[18] Y. Qiao, H. Wang, and D. Wang. Parallelizing and optimizing overlapping community detection with speaker-listener label propagation algorithm on multi-core architecture. In *2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 439–443, 2017. doi: 10.1109/ICCCBDA.2017.7951954.

[19] Z. Qin, Z. Cheng, C. Lin, Z. Lu, and L. Wang. Optimal workload allocation for edge computing network using application prediction. *Wireless Communications and Mobile Computing*, 2021(1):5520455, 2021. doi: https://doi.org/10.1155/2021/5520455. URL `https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/5520455`.

[20] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, page 507–516, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318693. doi: 10.1145/2433396.2433461. URL `https://doi.org/10.1145/2433396.2433461`.

[21] L. A. Wolsey. *Mixed Integer Programming*, pages 1–10. John Wiley Sons, Ltd, 2008. ISBN 9780470050118. doi: https://doi.org/10.1002/9780470050118.ecse244. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse244`.

[22] J. Xie, B. K. Szymanski, and X. Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *2011 IEEE 11th International Conference on Data Mining Workshops*, pages 344–349, 2011. doi: 10.1109/ICDMW.2011.154.

[23] Y. Yang, X. Du, Y. Ye, J. Ding, T. Wang, M. Chen, and K. Li. Multi-objective deep reinforcement learning for function offloading in serverless edge computing. *IEEE Transactions on Services Computing*, 18(1):288–301, 2025. doi: 10.1109/TSC.2024.3489443.

[24] L. Zhao, J. Liu, Y. Shi, W. Sun, and H. Guo. Optimal placement of virtual machines in mobile edge computing. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, 2017. doi: 10.1109/GLOCOM.2017.8254084.