



UNIVERSITY OF LEEDS

Optimisation of Service Placement in Multi-Access Edge Computing

Shuyi Chen

**Submitted in accordance with the requirements for the
degree of PhD. Computer Science**

**The University of Leeds
Faculty of Engineering and Physical Sciences
School of Computer Science**

Apr 2026

Intellectual Property Statement

I confirm that the work submitted is my own, except where work which has formed part of jointly authored publications has been included. My contribution and that of the other authors to this work have been explicitly indicated below. I confirm that appropriate credit has been given within the thesis where reference has been made to the work of others.

Some of the results and work presented in this thesis have been published in the following papers:

- Chapter 3 include material published in:
 - Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “Efficient Placement of Interdependent Services in Multi-access Edge Computing”. In: *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer. 2024, pp. 150–164.
- Chapter 4 include material published in:
 - Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “QoS-aware placement of interdependent services in energy-harvesting-enabled multi-access edge computing”. In: *Future Generation Computer Systems* (2025), p. 108009.
- Chapter 5 include material published in:

-
- Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “Dynamic Service Scheduling and Resource Management in Energy-Harvesting Multi-Access Edge Computing”. In: *2025 IEEE International Conferences on Internet of Things (iThings) IEEE Green Computing & Communications (GreenCom) IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE. 2025, pp. 122–130.

For the publications listed, I served as the primary investigator and lead author. I was independently responsible for the system modelling, algorithm design and implementation, execution of the experimental evaluations, and the drafting of all manuscripts.

Regarding co-author contributions, Nikos Tziritas provided high-level supervision and advisory discussions regarding the initial research direction. Panagiotis Oikonomou offered feedbacks on the result analysis and narrative structuring, alongside advisory recommendations for the dynamic migration policies. Zhengchang Hua provided supplementary technical assistance during the large-scale server experiments and aided in figure refinement. Nan Zhang contributed proofreading and editorial feedback on the drafted manuscripts. Finally, Georgios Theodoropoulos and Karim Djemame, as supervisors, secured funding and provided high-level guidance throughout the project.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement. The right of Shuyi Chen to be identified as author of this work has been asserted by Shuyi Chen in accordance with the Copyright, Designs and Patents Act 1988.

© 2026 The University of Leeds and Shuyi Chen

Acknowledgements

Throughout the process of completing this dissertation, I have been supported by many incredible individuals. I would like to express my deepest gratitude to all of them.

First and foremost, I would like to express my immense gratitude to my supervisors, Prof. Georgios Theodoropoulos and Prof. Karim Djemame. Their profound expertise, invaluable guidance, and continuous encouragement have been instrumental to my academic training and throughout my four-year journey. I am truly grateful for the academic freedom they provided and for their patience in guiding me through the complexities of my doctoral studies.

I would also like to thank Dr. Arash Bozorgchenani and Dr. Ning Zhang for serving as my examiners. I deeply appreciate the time and effort they dedicated to reviewing my work, and I will always cherish their valuable feedback, which has significantly strengthened this dissertation.

My sincere thanks go to Prof. Nikos Tziritas for his continuous guidance, as well as his inspiration and academic insights. I would also like to extend my thanks to Dr. Panagiotis Oikonomou for his generous feedback and suggestions during our collaborations. My appreciation also goes to Prof. Evangelos Pournaras and Prof. Zheng Wang for their invaluable suggestions during my Ph.D. transfer and annual reviews, helping me refine my research direction and confidently navigate the critical milestones of my degree.

For all the support and encouragement from my colleagues and friends at both the University of Leeds

and SUSTech, I express my profound thankfulness. Thank you to everyone in Lab 549b for creating such a warm and welcoming environment, where we could share our moments of happiness and successes while constantly absorbing new knowledge together. I also wish to thank Dr. Nan Zhang for his kind and constructive feedback. Thanks to everyone in the DSS Lab for the enjoyable and highly valuable discussions during our weekly reading groups and research talks. A special thanks to Dr. Abdullah Aljulyfi for his help surrounding simulators. These discussions have broadened my perspective and deeply enriched my understanding of this field.

To balance my work on this dissertation with the joys of life, I could always rely on my friends and family. I especially want to thank my STALY friends for keeping me motivated, focused, and relaxed when I needed it most. I would also like to express my appreciation to the Ms. Nagano; her work and characters cheered me up countless times. Furthermore, my deepest gratitude belongs to my parents for their unwavering love and support.

Last but certainly not least, I want to thank Zhengchang Hua for sailing through this journey with me, and for being an irreplaceable partner and my greatest supporter. Finally, a very special thanks to my Poochon sister, Maomao, for her fluffy and warm companionship, her endless tail wags, and her unconditional love.

Declaration of Generative AI Use

In accordance with university policy, I acknowledge the use of the following tools in the preparation of this thesis:

- Google Gemini 3.1: Used for content proofreading and grammatical corrections.
- Microsoft Copilot: Used for code auto-completion in experiment execution and result analysis.

Abstract

Multi-access edge computing, driven by the development of 5G communications, pushes network and computing resources closer to mobile users. By providing wireless access through base stations, various computation tasks can be offloaded from user equipment and executed on servers in nearby micro-datacentres. The proliferation of Internet of Things sensors, smart devices, and diverse applications leads to a surge of user-submitted queries that must be processed at the network edge, necessitating effective resource management.

In practice, many modern applications are not monolithic but are composed of connected components with distinct processing requirements and data dependencies. To better utilise limited edge resources and schedule these complex applications effectively, such task dependencies must be incorporated into resource allocation decisions.

For service providers, maintaining Quality of Service is a central goal. Beyond providing basic computing and storage, it is crucial to enhance the performance of data-intensive, latency-sensitive, and CPU-intensive applications. Key metrics reflecting Quality of Service include end-to-end latency, service throughput, scalability, and failure rate, all of which present significant challenges in dynamic edge computing environments.

This dissertation first addresses the challenge of scheduling dependent services. User queries are modelled as directed acyclic graphs to capture inter-task dependencies. Two novel heuristic-based

algorithms generate placement plans for these application modules. By identifying the critical path of the directed acyclic graph and evaluating server load alongside estimated latencies, the algorithms strategically assign modules to appropriate micro-datacentre servers, co-optimising end-to-end latency and server energy consumption.

The second part explores integrating energy harvesting technologies to establish a sustainable edge computing environment. To address limited and intermittent renewable energy in hybrid multi-access edge computing systems, an online algorithm dynamically manages edge server frequency scaling and cloud offloading. This strategy supplements local generation by offloading tasks to the cloud or purchasing grid power, promoting performance stability while reducing operational costs.

The third part tackles the instability of systems powered purely by harvested energy, where severe mismatches occur between volatile energy supplies and dynamic application demands. A novel online resource scheduling and energy management scheme is proposed to adaptively coordinate Dynamic Voltage and Frequency Scaling with horizontal service migration, maintaining stable performance in an edge system operating entirely on green energy.

The efficacy of the proposed algorithms and management schemes is validated through extensive simulation-based experiments against state-of-the-art baselines. The extended simulation framework is released as an open-source contribution. Results demonstrate that the developed algorithms significantly reduce energy consumption, minimise latency, and lower operational costs, contributing to an efficient and sustainable edge computing ecosystem.

Contents

1	Introduction	1
1.1	Research Background and Motivation	1
1.1.1	The Era of MEC	1
1.1.2	Resource Allocation for Interdependent Microservices	3
1.1.3	Green Computing and Energy Harvesting	4
1.2	The Core Problem	6
1.2.1	The Optimisation of Service Placement	6
1.2.2	Open Challenges	7
1.3	Research Questions and Objectives	8
1.4	Research Scope, Terminology, and Key Assumptions	9
1.4.1	Terminology and Conventions	9
1.4.2	Key Assumptions	10
1.5	Methodology	11
1.6	Main Contributions	11
1.7	List of Publications	13
1.8	Thesis Structure	13
2	Background	15

2.1	Introduction to MEC	16
2.1.1	Definition	16
2.1.2	Architecture	18
2.1.3	Paradigm Characteristics and Advantages	22
2.2	Edge Applications and Microservices	23
2.2.1	Applications in MEC	23
2.2.2	Application Models	24
2.3	Powering MEC Systems	26
2.3.1	Classification of Energy Sources	26
2.3.2	Power Management Techniques	27
2.4	The Service Placement Problem	29
2.4.1	Problem Definition	29
2.4.2	Constraints and Challenges	31
2.4.3	Placement Characteristics	32
2.5	Summary	34
3	Literature Review	36
3.1	Service Placement Optimisation	37
3.1.1	Optimisation Objectives	37
3.1.2	Methodological Approaches	40
3.2	State-of-the-Art in Dependency-Aware Service Placement	43
3.2.1	Formal Problem Formulations and System Models	44
3.2.2	Taxonomy and Analysis of Existing Placement Strategies	48
3.2.3	Literature Synthesis and Methodological Gaps	50
3.3	State-of-the-Art in Energy-Harvesting MEC Systems	53
3.3.1	Online Scheduling and Cloud Offloading	53

3.3.2	Power Management in Renewable Systems	55
3.4	Evaluation Environments and Tools	57
3.5	Critical Assessment and Research Gaps	60
3.6	Relation to Own Work	62
4	Dependent Service Placement in Cloud-MEC	64
4.1	Introduction	64
4.2	System Model and Formulations	66
4.2.1	Key Assumptions	66
4.2.2	MEC Network Model	67
4.2.3	Application Model	68
4.2.4	Problem Formulation	70
4.3	Algorithm Design	74
4.3.1	Energy-Aware Delay-Experienced Minimisation Algorithm	74
4.3.2	Delay-Aware Energy Minimisation Algorithm	78
4.4	Experimental Evaluation	82
4.4.1	Performance Indicators & Setup	82
4.4.2	Performance Assessment	83
4.5	Evaluation Results	85
4.6	Conclusion	92
5	Dynamic Scheduling in Energy-Harvesting MEC	93
5.1	Introduction	93
5.2	System Model and Formulations	94
5.2.1	Key Assumptions	94
5.2.2	MEC Network Model	95
5.2.3	Application Model	95

5.2.4	Problem Formulation	97
5.3	Algorithm Design	101
5.4	Experimental Evaluation	108
5.5	Evaluation Results	110
5.6	Conclusion	116
6	Scheduling under Harvest-Utility Mismatch	118
6.1	Introduction	118
6.2	System Model and Formulations	120
6.2.1	Key Assumptions	120
6.2.2	MEC Network Model	121
6.2.3	Application Model	121
6.2.4	Energy and Battery Model	122
6.2.5	Migration Model	123
6.2.6	Latency Model	124
6.2.7	Problem Formulation	124
6.2.8	System Assumptions and Applicability	129
6.3	Algorithm Design	129
6.3.1	Initial Assignment	131
6.3.2	Dynamic Resource Scheduling	131
6.3.3	Algorithm Complexity Analysis	139
6.4	Preliminary experiments	139
6.4.1	Performance Indicators & Setup	140
6.4.2	Ablation Study Results	143
6.5	Extended Experiments	147
6.5.1	Impact of DES Smoothing Factors	149

6.5.2	Impact of Migration Weight	151
6.5.3	Impact of Migration Overhead	152
6.5.4	Impact of Battery Safety Margin	153
6.5.5	Impact of Prediction Error	155
6.5.6	Summary	156
6.5.7	Discussion and Limitations	156
6.6	Conclusion	157
7	YAFS-MEC	159
7.1	The Standard YAFS Framework	159
7.2	Architectural Hierarchy and Topology	161
7.3	Service Dependency and Synchronisation	163
7.4	Resource Contention and Concurrent Task Processing	165
7.5	Energy Management and Stateful Monitoring	166
7.6	Summary of Simulator Contributions	167
8	Conclusion	168
8.1	Summary of Contributions	168
8.2	Limitations	169
8.3	Future Work	170
	References	172

List of Figures

1.1	The reference architecture of MEC integrated into 5G networks.	2
1.2	Dependent tasks in smart traffic management scenario.	3
1.3	High-level MEC ecosystem with energy harvesting.	5
1.4	Roadmap of the thesis.	14
2.1	The standardised MEC framework established by ETSI.	17
2.2	The MEC system architecture facilitating physical resource orchestration and task processing.	19
2.3	The decision flowchart of service request lifecycles in MEC.	21
2.4	Three different application deployment models in MEC.	25
2.5	A hybrid-powered MDC, supplied by both traditional grid and renewable sources. . .	27
2.6	The service placement process: mapping the service graph to the network graph. . .	30
2.7	The taxonomy tree of service placement characteristics.	32
4.1	An example Cloud-MEC network consisting of multiple MDCs and one cloud data centre, serving multiple end users.	68
4.2	An example application graph consisting of two data sources, six dependent services, and one end user.	69

4.3	The coarse-grain stage of EDEM: bottom-up critical path selection. The algorithm iteratively approximates the <i>EST</i> for each service on selected servers, designating the predecessor that maximises the minimum <i>EST</i> as the critical node at each hierarchical level. The final critical path is highlighted in red.	74
4.4	The coarse-grain stage of EDEM: service-to-MDC mapping. Red circles indicate critical modules along the identified critical path, whereas grey circles represent non-critical modules, with lighter shades denoting lower criticality. Black links indicate assigned placements between modules and target MDCs.	74
4.5	The fine-grain stage of EDEM: independent service-to-server mapping within each MDC.	75
4.6	The phase one of DEM: a level-order traversal of the application graph to assign modules layer-by-layer to edge servers pre-sorted by their energy consumption coefficients.	79
4.7	The phase two of DEM: iteratively reassign service modules to the least power-consuming servers across unexplored MDCs whenever energy consumption can be reduced.	79
4.8	The phase three of DEM: iteratively reassign critical-path modules to alternative servers and recalculating end-to-end latency until no further reductions are possible.	79
4.9	Normalised energy consumption (EC) across different network topologies with 20 MDCs and 4 servers per MDC.	87
4.10	Normalised user-experienced latency (LT) across different network topologies with 20 MDCs and 4 servers per MDC.	87
4.11	Normalised user-experienced latency (LT) with a varying number of MDCs ($n \in \{5, 10, 20\}$).	87
4.12	Normalised energy consumption (EC) with a varying number of MDCs ($n \in \{5, 10, 20\}$).	87
4.13	Normalised energy consumption (EC) with a varying number of servers per MDC ($m \in \{2, 4, 8\}$).	87

4.14	Normalised user-experienced latency (LT) with a varying number of servers per MDC ($m \in \{2, 4, 8\}$).	87
5.1	An example Cloud-MEC network with MDCs powered by energy harvesting devices.	96
5.2	Tasks arrivals and responses in the system.	96
5.3	An example of dependent modules forming an application.	97
5.4	Comparison of total processed requests (TP) for Job 3 and Job 10 across different scheduling policies.	110
5.5	Comparison of average user-experienced latency (LT) for Job 3 and Job 10 across different scheduling policies.	110
5.6	Comparison of the acceptance rate (AC) of processed requests for Job 3 and Job 10 across different scheduling policies.	111
5.7	Comparison of overall energy consumption (EC) at the edge for Job 3 and Job 10 across different scheduling policies.	111
5.8	Comparison of overall energy consumption (EC) at the cloud for Job 3 and Job 10 across different scheduling policies.	111
5.9	Comparison of cloud renting cost (CRC) for Job 3 and Job 10 across different schedul- ing policies.	112
6.1	Temporal mismatch between energy harvesting and consumption over a 24-hour period.	119
6.2	An example off-grid energy-harvesting MEC network with MDCs powered exclu- sively by renewable energy.	122
6.3	The workflow of the dynamic algorithm proposed.	130
6.4	The time-varying energy harvesting power and task arrival rate used to simulate the temporal mismatch between energy supply and user demand.	142
6.5	Impact of scaling the number of MDCs and servers on normalised latency (LT).	143
6.6	Impact of scaling the number of MDCs and servers on resource utilisation (RU).	144

6.7	Impact of the energy harvesting charging multiplier (ch) on normalised latency (LT) and throughput (TH).	145
6.8	Detailed performance of the SM algorithm in a case study with 2 MDCs and 4 servers per MDC ($ch = 1.25$).	146
6.9	Timeline of service migrations in the case study, showing most migrations occur during the low-energy period between 4000 and 8000 time steps.	147
6.10	Heatmap of service migration counts between servers in the case study, showing a balanced distribution and no "ping-pong" effects.	148
6.11	The dynamic energy harvesting and task arrival model used for the extended experiments, reflecting more realistic, fluctuating conditions.	149
6.12	Impact of different DES smoothing factor pairs (ω_a, ω_b) on average latency under increasing prediction noise levels.	150
6.13	Impact of different DES smoothing factor pairs (ω_a, ω_b) on task throughput under increasing prediction noise levels.	150
6.14	Impact of different DES smoothing factor pairs (ω_a, ω_b) on average energy consumption per task under increasing prediction noise levels.	151
6.15	Impact of the migration weight (ω_M) on the total number of migrations and the resulting average latency.	152
6.16	Impact of the migration overhead scaling factor (ω_O) on the total number of migrations for the proposed policy versus a migration-only baseline.	153
6.17	Impact of the battery safety margin (B_{safe}) on renewable energy utilisation and system throughput.	154
6.18	Robustness comparison of the algorithm with smoothed vs. non-smoothed inputs under increasing prediction error.	155
7.1	Core architecture and execution workflow of the YAFS framework.	160

7.2 Transmission of messages in the YAFS simulator. 161

7.3 The operational lifecycle of a service in the YAFS simulator. 162

7.4 Internal structure of a service featuring an extended local buffer. 163

7.5 The operational lifecycle of a service featuring dependency synchronisation. 164

7.6 Services operate as independent processes in the YAFS framework. 165

List of Tables

3.1	Summary of primary optimisation metrics and their mathematical definitions in the literature.	40
3.2	Comparison of dependency-aware placement algorithms in the literature.	52
4.1	List of key notations for the system model and placement algorithms.	66
4.2	MEC-cloud environment configurations.	84
4.3	Edge prioritisation results (<i>EP</i>).	90
4.4	Execution time in ms (<i>ET</i>).	90
5.1	Summary of key notations for the dynamic scheduling and offloading model.	102
5.2	Energy-harvesting MEC-cloud environment configurations.	109
6.1	Key notations for the energy management and task scheduling formulations.	130
6.2	Application characteristics.	141

List of Abbreviations

DAG	Directed Acyclic Graph
DEM	Delay-Aware Energy Minimisation
DES	Double Exponential Smoothing
DRL	Deep Reinforcement Learning
DSM	Dynamic Scaling and Migration
DSO	Dynamic Scaling and Offloading
DVFS	Dynamic Voltage and Frequency Scaling
EDEM	Energy-Aware Delay-Experienced Minimisation
EST	Earliest Start Time
ETSI	European Telecommunications Standards Institute
ILP	Integer Linear Programming
IoT	Internet of Things
LFT	Latest Finish Time
LLM	Large Language Model

LST	Latest Start Time
MDC	Micro-Datacentre
MEC	Multi-Access Edge Computing
MINLP	Mixed-Integer Non-Linear Programming
MOO	Multi-Objective Optimisation
QoS	Quality of Service
RAN	Radio Access Network
SFC	Service Function Chain
SLA	Service Level Agreement
UAV	Unmanned Aerial Vehicle
UE	User Equipment
VEC	Vehicle Edge Computing
VNF	Virtual Network Function
YAFS	Yet Another Fog Simulator

Chapter 1

Introduction

This chapter introduces the research context, motivation, and the core objectives of this dissertation. We first present the background of Multi-Access Edge Computing (MEC) and the critical need for fine-grained resource allocation, particularly in the context of complex, dependent services and energy harvesting systems. Following an analysis of the open challenges in current literature, we define our specific research questions and objectives. Finally, we outline the research methodology, detail the main contributions of this work, and provide a roadmap for the remainder of the thesis.

1.1 Research Background and Motivation

1.1.1 The Era of MEC

The world of MEC, driven by the rapid development of 5G communications, has grown from an early deployment concept into an essential pillar of today's mobile networks [96]. Mobile edge servers, located strategically near base stations, bring computing power closer to the user, significantly improving the accessibility of both information and resources. Figure 1.1 shows an MEC system in 5G networks, with multiple edge datacentres connected to the 5G New Radio base stations (gNBs),

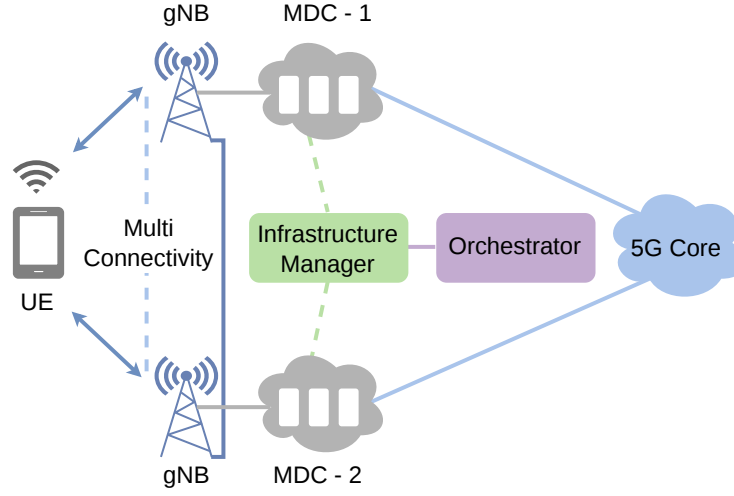


Figure 1.1: The reference architecture of MEC integrated into 5G networks.

serving the User Equipment (UE) with high-speed, low-latency communication.

In our daily lives, the proliferation of smart devices and the increasing complexity of applications dictate that vast numbers of mobile devices connect to the network from every corner of the globe. It is estimated that by 2030, there will be over 30 billion internet-connected devices, including smartphones and smart home devices [136].

The increasing ubiquity of smart mobile devices is driving a new wave of advanced applications, including e-health, smart manufacturing, autonomous driving, multiplayer mobile gaming, and augmented reality [6]. This hyper-connectivity leads to a massive surge of data that must be continuously transmitted and processed, up to 20 Gigabits per second [96]. This data comprises information gathered by sensors as well as diverse queries submitted by users. The queries range from latency-critical augmented reality overlays and real-time navigation adjustments to bandwidth-heavy video streaming and periodic Internet of Things (IoT) data uploads. Each query type presents distinct demands for computation, storage, and network resources. While these evolving technologies ease daily life and elevate network service quality, they simultaneously drive a relentless increase in overall computa-

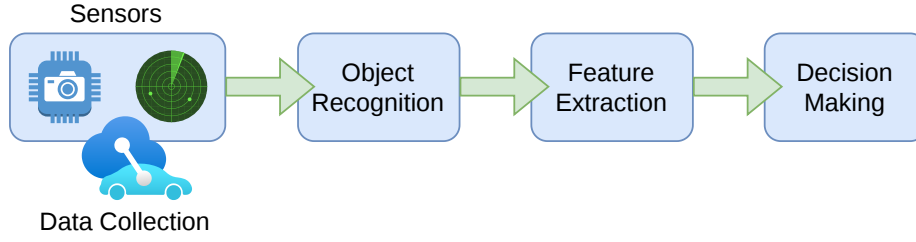


Figure 1.2: Dependent tasks in smart traffic management scenario.

tional demands.

The MEC paradigm effectively resolves the inherent resource constraints of most IoT and mobile devices, which typically suffer from limited processing power, storage capacity, and battery life [75]. By offloading tasks to edge resources, MEC seamlessly augments the operational capabilities of these end devices. Furthermore, the integration of MEC and 5G networks delivers substantial advantages for reducing latency and minimising operational costs [123].

1.1.2 Resource Allocation for Interdependent Microservices

In modern distributed computing systems, effectively assigning computation, communication, and storage resources is critical to meeting the intensive compute demands and stringent Service Level Agreements (SLAs) of user requests [87]. A well-designed allocation strategy helps maintain required performance levels while maximising the overall utilisation of available infrastructure.

In practice, many of these modern applications are no longer monolithic; instead, they are composed of a series of interdependent service components (or microservices) that can be individually deployed and managed [78]. Within these complex applications, dependencies among sub-components make task scheduling and resource management significantly more complicated, as execution often requires sequential inputs from sensors, databases, or preceding sub-tasks.

For example, as illustrated in Figure 1.2, in a smart city traffic management scenario, an applica-

tion designed to mitigate intersection accidents might need to fetch real-time video feeds, process them sequentially for object recognition, perform feature extraction to track vehicle trajectories, and aggregate this data into a core decision-making algorithm that controls traffic signals [1].

Compared to treating the application as a single, indivisible task, implementing a fine-grained resource allocation strategy for each sub-component utilises distributed edge resources far more effectively. Distributing these service components allows multiple edge servers to serve users collaboratively [112]. However, it also introduces new challenges in satisfying user demands and improving overall performance metrics, such as end-to-end latency.

1.1.3 Green Computing and Energy Harvesting

Concurrent with the pursuit of utilising computational and network resources to their fullest extent, the growing trend of green computing has made energy conservation a paramount concern.

As computing demand grows, so do electricity consumption and greenhouse gas emissions. Sustainable, or green, computing focuses on the design, deployment, and management of infrastructure to maximise energy efficiency and minimise carbon emissions [4]. Enhancing the environmental sustainability of these systems is crucial for mitigating the ecological consequences of global digitalisation and achieving long-term operational cost reductions.

Recent work highlights that for many end devices, the execution of edge applications accounts for the majority of their total carbon footprint [131]. Therefore, renewable energy is increasingly being adopted to reduce computing's carbon footprint. Energy harvested from ambient resources, such as solar and wind, can support the sustained, low-carbon operation of computing systems. In contrast to conventional grid (or "brown") energy, which is costly and generates a significant carbon footprint, energy harvesting enables service deployment in remote areas where grid power is inaccessible.

Integrating energy harvesting into the MEC environment offers a highly promising approach to build-

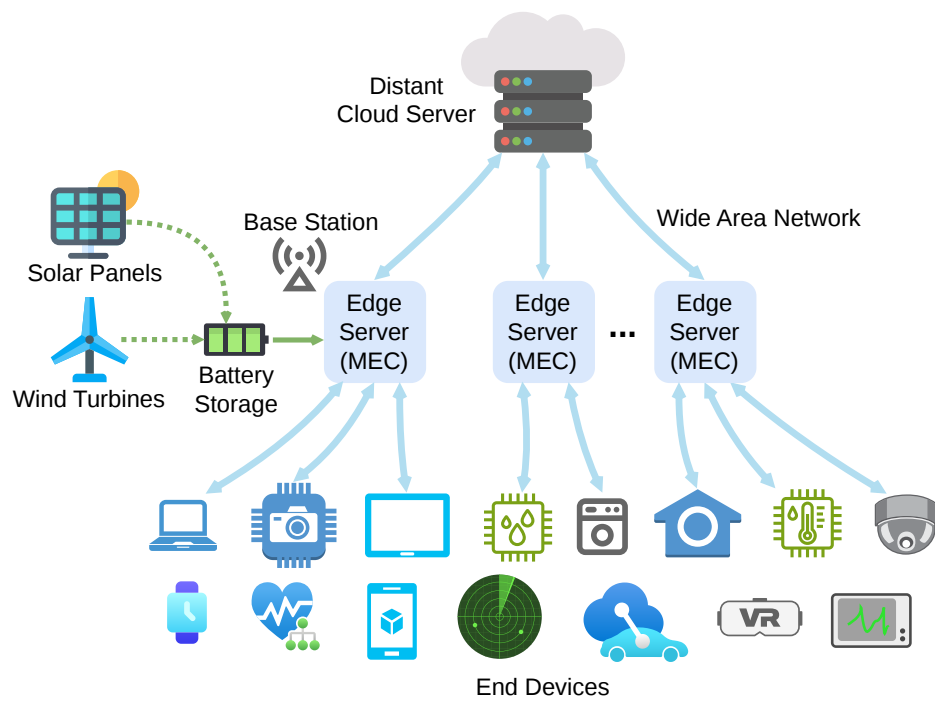


Figure 1.3: High-level MEC ecosystem with energy harvesting.

ing sustainable networks. As illustrated in Figure 1.3, one example involves deploying local renewable sources paired with rechargeable batteries to power edge servers. Practical implementations of this paradigm include self-powered industrial monitoring systems [164] and solar-powered smart city nodes [3] that process data autonomously at the network edge. However, adopting these techniques introduces new challenges due to the dynamic nature of user requests, coupled with the unpredictability and diversity of green energy supplies [47].

1.2 The Core Problem

This dissertation investigates the service placement problem within MEC environments, with a primary focus on optimising resource allocation and job scheduling.

1.2.1 The Optimisation of Service Placement

Service placement is a fundamental resource allocation mechanism that determines exactly where and how computational tasks, applications, or individual service components are deployed across available edge computing servers and central clouds. Technically, a *service* represents a modular software entity, such as a microservice or a Virtual Network Function (VNF), encapsulated within a lightweight virtual machine or container. In modern complex edge applications, services do not operate in isolation; rather, an application is composed of multiple interconnected VNFs structured as a Directed Acyclic Graph (DAG) or a Service Function Chain (SFC). Each constituent VNF requires a specific volume of computational instructions (CPU cycles) and memory footprint, while dependencies between parent and child VNFs entail transferring precise volumes of intermediate data across network links. Formally, the service placement problem is defined as a combinatorial mapping optimisation challenge. Given a dynamic set of incoming user requests triggering complex DAG applications and an MEC environment containing multiple micro-datacentres (MDCs), where each MDC maintains several edge computing servers with varying computation capacities, memory limits,

and time-varying power states, the placement problem seeks to find the optimal binary assignment mapping of each VNF to a specific hosting server. Concurrently, it determines the optimal network routing paths for data dependencies between distributed servers. Optimising this mapping is critical because edge infrastructure is inherently resource-constrained and heterogeneous [87]. Arbitrary or static placement leads to severe resource fragmentation, server overloading, bottlenecks, and degraded end-to-end latency. By intelligently optimising service placement, the system balances competing objectives [120], achieving low end-to-end latency for time-critical applications while minimising overall energy consumption and operational costs across the infrastructure.

1.2.2 Open Challenges

When tackling service placement optimisation across these diverse objectives, several critical gaps remain to be addressed:

Conflict Between Latency and Energy in Complex Applications: Optimisation goals frequently present conflicting trade-offs [162]. For instance, placing an object detection module near a security camera reduces communication delays but may inflate execution time if the local edge device possesses limited computational power [43]. Conversely, offloading the task to a more powerful, distant server reduces processing time but spikes energy consumption and transmission latency. Furthermore, existing co-optimisation methods typically treat applications as atomic units, largely overlooking the complex inter-dependencies and data flows between individual microservices [133].

Stochastic Energy Supply vs. Strict Service Deadlines: In energy-harvesting MEC systems, the energy supply is inherently limited, intermittent, and stochastic [79]. When local harvested energy is insufficient, the system faces a critical decision: purchase grid energy to process the task locally, or offload the task to the cloud. Formulating this decision policy while strictly adhering to the SLAs and stringent deadlines of time-sensitive applications remains a significant open challenge [11].

Time-Varying Workloads and Supply-Demand Mismatches: In real-world scenarios, user de-

mand fluctuates significantly throughout the day, creating distinct peak and valley usage hours [72]. Concurrently, energy generation (especially from renewable sources) is dictated by unpredictable environmental factors, leading to frequent supply-demand mismatches [32]. There is a notable lack of contemporary research addressing dependency-aware, fine-grained service placement within such dynamic, energy-constrained environments. Specifically, more work is needed on the safe and efficient utilisation of surplus battery storage alongside techniques like Dynamic Voltage and Frequency Scaling (DVFS) to adapt to these continuous fluctuations [34].

1.3 Research Questions and Objectives

The overarching objective of this dissertation is to propose resource management mechanisms for MEC systems to optimise the trade-off between energy efficiency and Quality of Service (QoS) performance for complex applications under dynamic and stochastic conditions.

To achieve this overarching goal, the research is driven by three specific objectives, each addressing a core research question:

Objective 1: To formulate dependency-aware service placement algorithms that minimise both end-to-end latency and server energy consumption for microservice architectures.

- **RQ1:** How can we design efficient, dependency-aware service placement algorithms that co-optimise end-to-end latency and server energy consumption for complex microservice applications?

Objective 2: To develop adaptive, online scheduling strategies for energy-harvesting-powered MEC environments that minimise operational costs while adhering to strict application deadlines.

- **RQ2:** In energy-harvesting MEC environments, how can online scheduling strategies be designed to adaptively manage cloud offloading and server scaling to minimise the operational

cost without violating strict application deadlines?

Objective 3: To devise dynamic resource scheduling and power-saving mechanisms that mitigate service disruptions and manage battery depletion in systems powered entirely by renewable energy.

- **RQ3:** How can dynamic resource scheduling and power-saving techniques (such as DVFS) be leveraged to safely manage battery depletion and supply-demand mismatches in systems powered entirely by renewable energy?

1.4 Research Scope, Terminology, and Key Assumptions

This research is situated within the domain of resource allocation for MEC systems. The scope is specifically constrained to the algorithmic design and systemic scheduling of computational and energy resources. Physical-layer radio transmission protocols and the low-level hardware design of energy harvesting circuits are explicitly excluded from this study.

1.4.1 Terminology and Conventions

To establish clear, consistent terminology throughout this dissertation, we define the following conventions for key terms and concepts:

- **Paradigm and System Architecture:** The term *edge infrastructure* is used as a generalised abstraction denoting the collective physical, virtual, compute, storage, and networking resources deployed along the continuum between end-user devices and the centralised cloud core. Under this broader umbrella, two distinct architectural paradigms are referenced:
 - *MEC (or Cloud-MEC System/Environment):* The telecom-centric edge paradigm that integrates distributed compute resources directly into cellular operator infrastructure (e.g., base stations) to provide low-latency processing near the access network.

- *Fog Computing*: A decentralised, multi-layered system architecture that extends compute, storage, and network management across heterogeneous intermediate devices (e.g., local gateways, routers, and edge nodes) spanning the entire path from end devices to the central cloud, independent of cellular network boundaries.
- **Compute Entities**: A *MDC* represents a small-scale, localised edge facility deployed at the network edge (e.g., at a base station). An MDC houses one or more physical *edge servers* (compute nodes). Remote cloud computing facilities are designated as *cloud datacentres* or *cloud servers*.
- **Client Devices**: *UE* (e.g., smartphones, IoT sensors, autonomous vehicles) denotes end-user client devices that generate application requests and offload computation tasks to edge or cloud servers.
- **Temporal Granularity**: In dynamic online scheduling frameworks (such as Markovian decision processes), continuous time is discretised into sequential *time slots* (denoted by t). The term *time step* is used synonymously in discrete-event simulation contexts.

1.4.2 Key Assumptions

To formulate our system model, we assume that the dynamic energy consumed by the edge server's CPU during task processing dictates the system's overall energy footprint. Consequently, our optimisation framework focuses exclusively on dynamic computational power. Our scheduling model restricts its scope to computation-bound applications characterised by high CPU-cycle demands and low input/output data payloads. Under these strict parameters, we treat static power dissipation and network-induced energy costs, such as communication and transmission overhead, as negligible relative to active computational energy consumption.

1.5 Methodology

To address the research questions, this research employs a combination of mathematical modelling, algorithmic design, and extensive system simulation:

Mathematical Modelling: Formulating the service placement and energy management challenges as constrained optimisation problems, including Multi-Objective Optimisation (MOO) and Mixed-Integer Non-Linear Programming (MINLP).

Algorithm Design: Developing low-overhead heuristic algorithms for the NP-hard placement problem, integrated with online scheduling mechanisms to adapt to stochastic energy arrivals and mitigate the risk of node depletion and subsequent QoS violations.

Simulation and Validation: Designing and implementing a custom simulation toolkit by extending the Yet Another Fog Simulator (YAFS) [65] framework. This enhanced simulator explicitly models MEC architectures, complex internal service dependencies, and incorporates real-world data traces for both user workloads and environmental energy generation to validate the proposed strategies.

While empirical evaluation on physical testbeds is a recognised approach, deploying a full-scale, real-world MEC environment is complex and economically impractical. This logistical challenge is magnified when powering these distributed nodes via energy harvesting devices, as physical weather conditions and ambient energy generation rates cannot be deterministically controlled to establish fair baseline comparisons. Therefore, simulation was selected as the most robust and rigorous methodology for this research.

1.6 Main Contributions

This dissertation advances the current state-of-the-art by addressing the complex interplay between maintaining QoS and enhancing energy efficiency. The specific contributions are as follows:

1. **Dependency-Aware Heuristic Algorithms: (Addressing RQ1)** We develop and experimentally evaluate two novel algorithms, Energy-Aware Delay-Experienced Minimisation (EDEM) and Delay-Aware Energy Minimisation (DEM), for the service placement problem in MEC. These heuristics are explicitly designed to simultaneously reduce end-to-end latency and server energy consumption for complex, time-sensitive applications.
2. **Enhanced Simulation Toolkit: (Detailed in Chapter 7)** We design and implement a comprehensive simulation platform by extending the standard YAFS simulator. While native YAFS models basic event scheduling and simple topologies, our extensions incorporate essential MEC capabilities: explicit hierarchical modelling of multi-server MDCs, integration of time-varying energy harvesting models and green battery dynamics, DVFS execution modes, live state and container migration modelling, and the ingestion of real-world industrial datasets (e.g., Alibaba cluster workloads and meteorological solar/wind traces) for rigorous validation of dependency-aware placement strategies.
3. **Online Scheduling for energy-harvesting MEC Systems: (Addressing RQ2)** We propose a novel online scheduling strategy, Dynamic Scaling and Offloading (DSO), for energy-harvesting-powered MEC systems. This algorithm adaptively manages dynamic server scaling and selective cloud offloading for dependency-aware services, co-optimising operational cost and QoS using real-world data traces.
4. **Depletion-Safe Energy Management Scheme: (Addressing RQ3)** We propose and evaluate a dynamic resource scheduling and energy management scheme, Dynamic Scaling and Migration (DSM), that addresses the challenge of battery depletion safety in MEC systems powered solely by renewable energy, operating effectively under highly dynamic supply and demand conditions.

1.7 List of Publications

- Shuyi Chen et al. “Efficient Placement of Interdependent Services in Multi-access Edge Computing”. In: *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer. 2024, pp. 150–164
- Shuyi Chen et al. “QoS-aware placement of interdependent services in energy-harvesting-enabled multi-access edge computing”. In: *Future Generation Computer Systems* (2025), p. 108009
- Shuyi Chen et al. “Dynamic Service Scheduling and Resource Management in Energy-Harvesting Multi-Access Edge Computing”. In: *2025 IEEE International Conferences on Internet of Things (iThings) IEEE Green Computing & Communications (GreenCom) IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE. 2025, pp. 122–130

1.8 Thesis Structure

The remainder of this dissertation is organised as follows, with the structural roadmap illustrated in Figure 1.4.

Chapter 2 establishes the fundamental concepts, architecture, characteristics, application scenarios, and power mechanisms of MEC.

Chapter 3 details relevant, state-of-the-art literature regarding service placement, scenario characteristics, operations under limited resources, utilised methodologies, optimisation objectives, and evaluation testbeds.

Chapter 4 introduces two novel service placement algorithms designed specifically for dependent services within MEC environments.

Chapter 5 proposes a dynamic service offloading and resource scheduling strategy for energy-harvesting

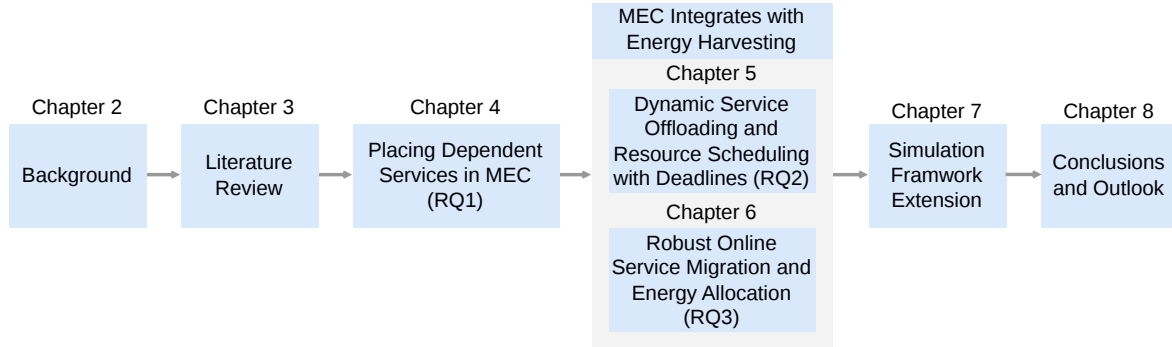


Figure 1.4: Roadmap of the thesis.

MEC systems handling deadline-constrained tasks.

Chapter 6 presents a robust, depletion-safe online strategy that jointly manages service scheduling and energy allocation for complex applications on purely energy-harvesting MEC systems.

Chapter 7 details the YAFS-MEC simulation framework, outlining the architectural and energy-aware extensions engineered to validate resource orchestration within energy-harvesting edge environments.

Chapter 8 summarises the addressed challenges, concludes the dissertation, and discusses implications for future research directions.

Chapter 2

Background

This chapter establishes the foundational concepts and theoretical background of this PhD project, exploring the prevailing opportunities and challenges within the domain.

Section 2.1 provides a comprehensive overview of MEC, detailing its definition and architecture (Sections 2.1.1 and 2.1.2), alongside its core characteristics and advantages (Section 2.1.3). Section 2.2 shifts the focus to edge applications and microservices, outlining modern use cases suited for MEC (Section 2.2.1) and presenting common application models (Section 2.2.2).

The discussion then moves to the energy dynamics of MEC systems in Section 2.3. Here, we review primary energy supply categories (Section 2.3.1) and evaluate standard power management techniques (Section 2.3.2).

Finally, Section 2.4 addresses the service placement problem. After establishing a formal definition (Section 2.4.1), we analyse the typical barriers and pitfalls of placing services in an MEC environment. The chapter concludes with Section 2.4.3, which proposes a comprehensive taxonomy of the placement problem based on its distinct characteristics.

2.1 Introduction to MEC

In this section, we provide a brief overview of MEC, covering its definition, architecture, main characteristics, advantages, applications, and power supply considerations.

2.1.1 Definition

Driven by the proliferation of fifth-generation networks, MEC has emerged as a vital solution that extends cloud computing capabilities directly to the edge of the network. The concept was formally initiated in 2014 by the European Telecommunications Standards Institute (ETSI) MEC Industry Specification Group [54].

ETSI established the standardised MEC framework, as illustrated in Figure 2.1 (adapted from the framework in ETSI GS MEC 003[39]). The core entities within this framework are categorised into three fundamental levels: the system level, the host level, and the network level. This architecture facilitates the deployment of MEC applications as software-only entities operating atop a virtualisation infrastructure situated at or near the network edge.

At the system level, the management framework provides a centralised, high-level overview of the entire edge ecosystem. It coordinates multiple MEC hosts to manage resource provisioning, scheduling, and overall application performance.

At the host level, the MEC platform and virtualisation infrastructure supply the necessary compute, storage, and network resources to accommodate and execute applications and services. The MEC platform itself comprises a collection of essential functionalities required to run MEC applications, enabling them to both consume and provide MEC services. These applications are deployed at the host level based on configurations or instantiation requests validated by the system-level management.

Finally, the network level encompasses entities such as 3GPP cellular networks, local area networks, and external public networks. These entities provide the essential connectivity and data plane routing

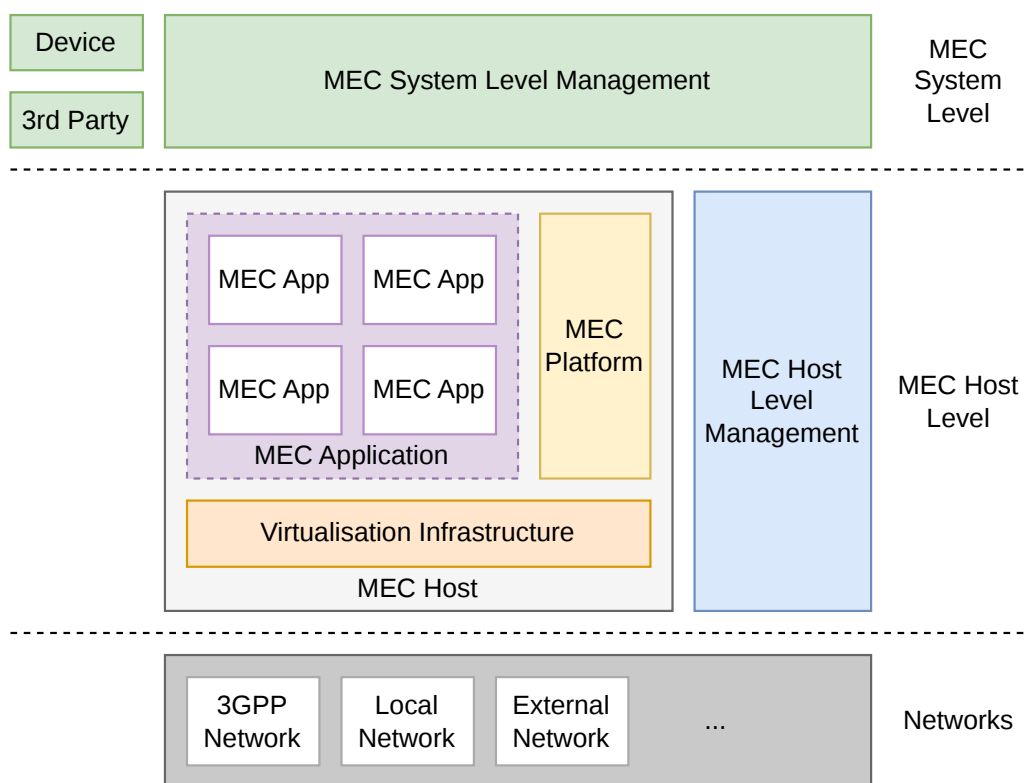


Figure 2.1: The standardised MEC framework established by ETSI.

required to seamlessly link UEs with MEC hosts, while also managing traffic steering between the edge infrastructure and the broader Internet.

Characterised by ultra-low latency, high bandwidth, and real-time access to radio network information, MEC offers both mobile users and content providers a robust information technology service environment and cloud-computing capabilities directly at the radio access network (RAN) edge [33].

Within this ecosystem, network operators can facilitate the flexible and rapid deployment of third-party applications and services directly to mobile subscribers. Consequently, MEC empowers a wide variety of time-sensitive and data-intensive applications, ranging from interactive mobile gaming, real-time video analytics, and connected vehicle communications [85], to industrial IoT data processing, remote healthcare, financial services, and immersive augmented reality experiences [129].

2.1.2 Architecture

Following the ETSI reference framework established in Section 2.1.1, which defines the logical entities and functional requirements of MEC, we introduce the hierarchical system architecture utilised to facilitate physical resource orchestration and task processing.

As illustrated in Figure 2.2, the system comprises an application layer, an underlying infrastructure layer, and a MEC control plane situated between the two.

User applications, including augmented reality/virtual reality interfaces, healthcare monitoring systems, and smart home controllers, reside at the application layer and produce a continuous influx of service requests demanding computational resources.

The control plane maps directly to ETSI's system-level management, with the MEC Orchestrator serving as its central coordinating entity. The Orchestrator maintains a comprehensive view of the entire system topology, monitoring edge hosts, available resources, and active services. It onboards application packages, effectively managing resource allocation and deploying services across the edge

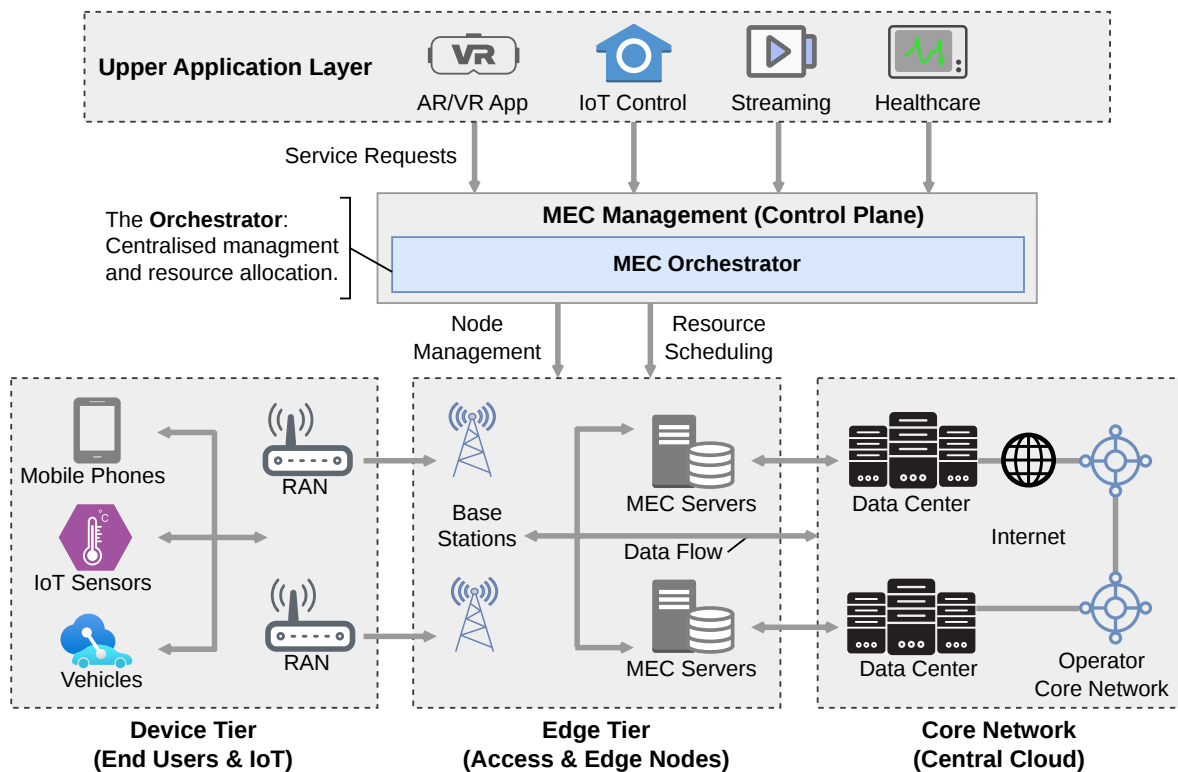


Figure 2.2: The MEC system architecture facilitating physical resource orchestration and task processing.

computing environment. Furthermore, it handles application instantiation and termination, and triggers dynamic application relocation when supported and necessary [62].

The underlying infrastructure layer is composed of three distinct tiers: the device tier, the edge tier, and the core network tier.

End-user mobile devices and various IoT devices, such as surveillance cameras, smartwatches, and environmental sensors, are geographically distributed within the device tier and integrate seamlessly into this architecture.

The edge tier, which realises ETSI's host-level management, contains the edge servers. These servers are typically located at the RAN edge, often physically co-located with or directly connected to base stations. A foundational component of the MEC host platform is the MDC, which acts as a small-scale, localised data centre deployed at the edge of the mobile network [115]. MDCs provide critical computing and storage resources in close physical proximity to the end user. The distributed edge servers are housed and managed within these respective MDCs.

Finally, the core network tier represents traditional, centralised cloud data centres and the broader core network infrastructure.

To illustrate the lifecycle of a service request within this architecture, consider a computation-intensive and highly latency-sensitive task, such as a mobile user engaged in an immersive augmented reality application that requires real-time object recognition. As shown in Figure 2.3, the request originates at the device tier and is transmitted to the system via the RAN.

Upon arrival, the request and its specific QoS constraints (e.g., strict maximum latency thresholds) are intercepted and evaluated by the MEC Orchestrator in the control plane. The Orchestrator queries the topology and current status of the edge tier, evaluating the available computing capacity, memory allocation, and network bandwidth of the local MDCs.

If a local MDC possesses sufficient resources, the Orchestrator schedules the task for immediate

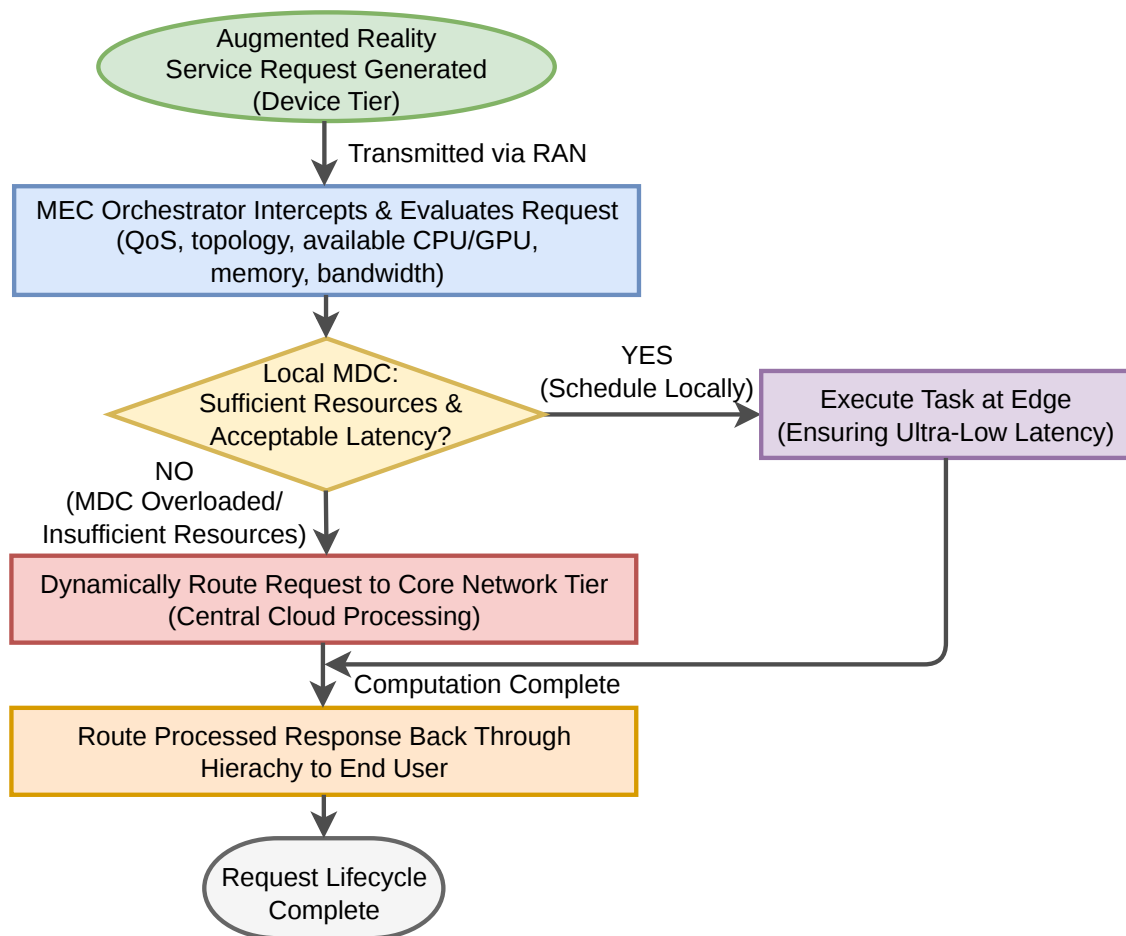


Figure 2.3: The decision flowchart of service request lifecycles in MEC.

execution at the edge, mitigating latency to maintain a responsive augmented reality experience. Conversely, the Orchestrator may determine that the edge is overloaded due to a lack of available CPU/GPU cycles or sufficient memory at the local MDC. In this scenario, it dynamically routes the request to the core network tier for processing in the central cloud. Once the computation is complete, the response is efficiently routed back through the hierarchy to the end user.

2.1.3 Paradigm Characteristics and Advantages

The MEC paradigm involves a number of distinct characteristics and advantages. We outline the main ones below:

Proximity: The physical proximity of computing resources to the end-users is a defining characteristic of MEC [33]. By deploying MDCs at the RAN edge, often collocated with the base station, MEC minimises the physical distance data must travel.

Ultra-low Latency: Many latency-sensitive applications require ultra-low latencies. According to [109], some applications may require processing times in the range of single-digit milliseconds. As a direct result of proximity, tasks do not necessarily require routing across long network paths to centralised cloud data centres. Instead, executing them on the local MEC host allows for significantly lower latency, which is unachievable with centralised cloud architectures [51].

Location Awareness: As an integral part of the wireless network infrastructure, MEC hosts possess inherent knowledge of the locations of connected user devices. This allows service providers to adjust resource allocation plans based on granular location data without relying on external global positioning system source data [20].

Scalability: As the number of smart devices proliferates and generates massive amounts of data, processing all tasks in the centralised Cloud is neither efficient nor feasible. MEC serves as a complementary paradigm to support increasing user requests, thereby aiding the scalability of such systems

[60].

Enhanced Quality of Experience: The combination of low latency and high bandwidth results in a seamless and highly responsive experience for end-users. This leads to enhanced Quality of Experience [129], particularly for interactive applications such as online gaming and augmented reality.

Context-Aware Service Adaptation: By accessing real-time network condition information, MEC enables intelligent services that can react to the environment and dynamically adapt their behaviour to maintain performance [58].

2.2 Edge Applications and Microservices

Having established the physical system architecture and resource orchestration mechanisms that underpin the MEC framework, we examine the software workloads that operate within this environment. This section explores the diverse use cases that necessitate edge deployment, and details the structural paradigms used to characterise these workloads, tracing the evolution from traditional monolithic deployments to modern, interconnected services.

2.2.1 Applications in MEC

As MEC systems increasingly permeate various sectors, they address a diverse range of computational challenges. Nowadays, MEC is envisioned for various critical areas, such as the following:

Internet of Medical Things: Uses wearable devices to provide continuous healthcare monitoring and remote diagnostic services [45].

Immersive Media (Augmented Reality and Virtual Reality): Immerses users in digital environments or overlays critical information onto the physical world, aiding in complex tasks like surgery, professional training, and simulation [38].

Autonomous Systems (Internet of Vehicles and Self-Driving): MEC integrates connected vehicles with regional traffic infrastructure, enabling the low-latency data processing required to enhance navigational safety and improve dynamic routing [161].

Smart Surveillance: Employs smart cameras to recognise faces and objects in real time, enhancing public security and improving traffic management [113].

Edge Intelligence and Large Language Models (LLMs): Deploys LLMs and artificial intelligence inference directly at the edge. While this reduces latency and protects data privacy, it requires rigorous resource scheduling to handle the heavy computational workloads [110].

These use cases are typically considered too computationally intensive to be executed solely on UE, yet too latency-sensitive to be offloaded to a distant, centralised cloud. Deploying these services on MEC hosts directly improves end-user experiences. To maximise computation resource utilisation at the edge, these services are most often delivered as virtual machine images or containerised microservices. Because these architectures generate complex service dependencies, intelligent resource orchestration is essential to maintain system stability and meet strict latency requirements.

2.2.2 Application Models

Several abstractions and model definitions are used in the literature to characterise the applications running on the MEC platform. Here, we briefly describe the main categories of these application models, as illustrated in Figure 2.4.

Monolithic Service: In this model, the user-defined application is offloaded as a single, holistic component. A common example is a legacy image processing task where the entire workflow is bundled into a single Virtual Machine image that must be executed as one atomic unit [77]. This unified workflow includes loading the image, applying filters, and compressing the output.

Set of Services with No Dependencies: In this scenario (illustrated in Figure 2.4(b)), the application

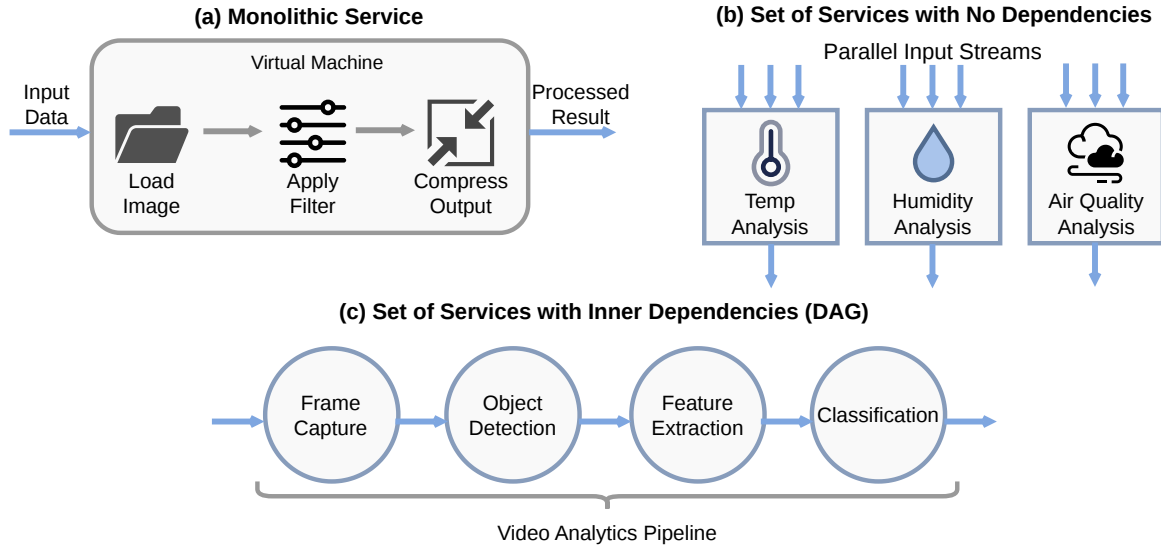


Figure 2.4: Three different application deployment models in MEC.

is pre-partitioned into a set of distinct components, each performing a specific functionality. Unlike sequential workflows where data must flow from one module to the next, there are no execution dependencies between these components; they operate completely in parallel. A typical example is an IoT environmental monitoring system where distinct data streams (e.g., temperature, humidity, and air quality) are offloaded to separate analysis functions that process the data independently and simultaneously [147].

Set of Services with Inner Dependencies: This type of application is composed of a set of inter-dependent components (microservices) and is typically represented as a connected graph. The edges represent the flow of data, and the vertices represent the processing components. The DAG model is frequently used to model such applications. Practical and realistic examples include Video Analytics pipelines (e.g., for face recognition) [145]. Such an application typically consists of a *Frame Capture* service, which passes data to an *Object Detection* service, which in turn feeds into a *Feature Extraction* service, and finally a *Classification* service. The execution of each downstream component is strictly blocked until it receives input from its predecessor.

2.3 Powering MEC Systems

As discussed in the previous section, deploying diverse edge applications and microservice architectures introduces computational demands at the network periphery that are highly variable and often intensive. Moreover, transitioning to highly granular, interdependent service models like the DAG-based pipelines mentioned earlier unlocks immense application flexibility. While the software models enabling these distributed workloads are widely adopted, sustaining the physical infrastructure required to execute them presents an entirely different set of challenges. Specifically, the reliable and efficient provision of electrical energy remains a critical open issue in MEC systems [116]. This section establishes the energetic constraints of MEC environments, categorising power sources and their associated management techniques.

2.3.1 Classification of Energy Sources

The operational sustainability and geographical feasibility of a MEC deployment depend heavily on its underlying energy infrastructure. Unlike centralised cloud data centres, which are typically situated in areas with robust, industrial-grade power availability, edge servers are highly distributed. They must operate in heterogeneous environments ranging from dense urban hubs to remote, off-grid locations. Consequently, MEC system architects must leverage different energy provisioning strategies tailored to specific deployment constraints, budgetary limits, and environmental sustainability goals [93]. The energy supplies utilised to power these systems generally fall into two primary categories, each presenting distinct advantages and limitations:

Grid Energy: Powering MEC hosts via the traditional power grid is currently the most common and stable approach. By consuming grid power, service providers can maintain continuous availability and support high-performance computing workloads without the constraints of battery depletion [37]. However, reliance on brown energy contributes to a larger carbon footprint and carries significant environmental impacts [10]. Furthermore, in remote deployment scenarios, such as rural areas or

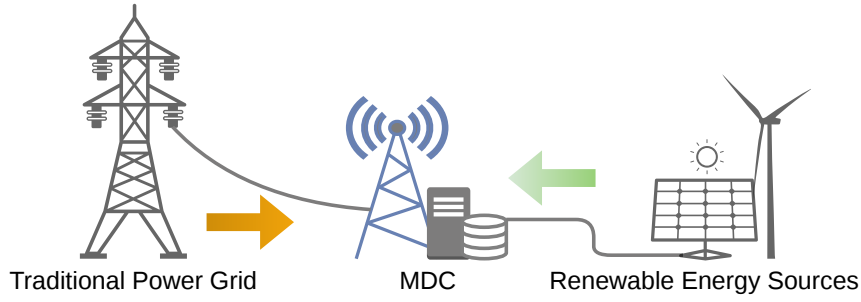


Figure 2.5: A hybrid-powered MDC, supplied by both traditional grid and renewable sources.

difficult terrains, establishing a connection to the wired grid may not be a feasible option due to high deployment costs [36].

Renewable Energy: Renewable energy sources, including solar, wind, and kinetic energy, have been increasingly applied to power edge devices and deployed near base stations in modern MEC systems [119]. By utilising energy harvesting techniques, the operation of MEC hosts can be supported by clean, green energy, enabling MEC platforms to function effectively in remote, off-grid areas [105]. However, due to the inherent intermittent nature of such energy sources, environmental conditions have a profound impact on energy generation, rendering it less reliable and predictable than the traditional grid.

Figure 2.5 illustrates an MDC connected to both a conventional utility grid and an energy harvesting module. This hybrid architecture enables the system to draw from both grid power and renewable energy sources, providing both energy efficiency and self-sustainability [28].

2.3.2 Power Management Techniques

As computing paradigms scale across environments ranging from hyperscale cloud data centres to resource-constrained IoT networks, the imperative for robust energy conservation and power management has become a universally critical challenge [92]. To mitigate energy consumption and reduce power wastage, a variety of methods have been proposed in the literature and adopted by modern

platforms, such as enterprise cloud environments (e.g., Amazon Web Services, Google Cloud) and contemporary mobile architectures [104]. We briefly summarise the most prevalent power management approaches below.

DVFS: This technique dynamically adjusts a processor's operating voltage and clock frequency based on real-time workload demands. By allowing processors to operate at lower voltage levels during periods of light utilisation, DVFS minimises dynamic energy consumption while maintaining active task execution [106].

Task and Virtual Machine Consolidation: This technique frames resource allocation as a multi-dimensional bin-packing problem, dynamically migrating virtual machines to tightly aggregate workloads and minimise the active set of physical host servers. By maximising resource utilisation on active nodes, consolidation allows redundant physical machines to be powered down, significantly reducing the aggregate baseline energy footprint of the computing cluster [17].

Dynamic Power Management and Sleep States: This strategy involves dynamically toggling the power states of hardware components or entire servers based on current user demand [29]. When idle periods are detected, the system transitions inactive servers into low-power sleep states, which directly curbs the data centre's static power consumption.

Computation Offloading: Commonly utilised in mobile and edge computing, this approach transfers computationally intensive workloads from resource-constrained client devices to more capable edge servers or remote cloud servers. Unloading these complex tasks reduces the local processing burden and extends the battery life of end-user devices [35], balancing the overall energy distribution across the network.

Despite the proven efficacy of these techniques, system architects must carefully navigate the inherent trade-off between energy savings and QoS or task execution performance [80]. Decreasing processor voltage and frequency via DVFS inevitably causes execution slowdowns; migrating virtual machines

or offloading tasks introduces network latency and migration overhead; and transitioning servers in and out of sleep states incurs both wake-up delays and transient energy spikes [30]. Therefore, an optimal power management policy must intelligently balance energy conservation goals with strict performance constraints.

2.4 The Service Placement Problem

To navigate the delicate trade-offs between energy conservation and strict QoS constraints established in Section 2.3, system architects require sophisticated resource orchestration. Within the MEC paradigm, this orchestration is formulated as the service placement problem.

The service placement problem has been extensively investigated in the literature, leading to the emergence of diverse research branches. Depending on the specific application scenarios and optimisation goals, various service placement solutions exist. In this section, we provide a comprehensive overview of the following aspects: problem definition, inherent constraints and challenges, problem taxonomy based on key characteristics. These aspects allow us to provide a holistic description of service placement within the MEC paradigm.

2.4.1 Problem Definition

The service placement problem fundamentally defines a mapping scheme, wherein logical service components (depicted on the left of Figure 2.6 as an application graph) are assigned to physical infrastructure resources (depicted on the right of Figure 2.6 as a physical network graph) [120]. In the physical network graph on the right, the vertices represent computing nodes, specifically edge computing servers housed within distributed MDCs alongside central cloud servers, while the interconnecting edges represent the physical communication links between them. Depending on the deployment environment, the connection between two edge servers can be realised either via high-speed wired optical

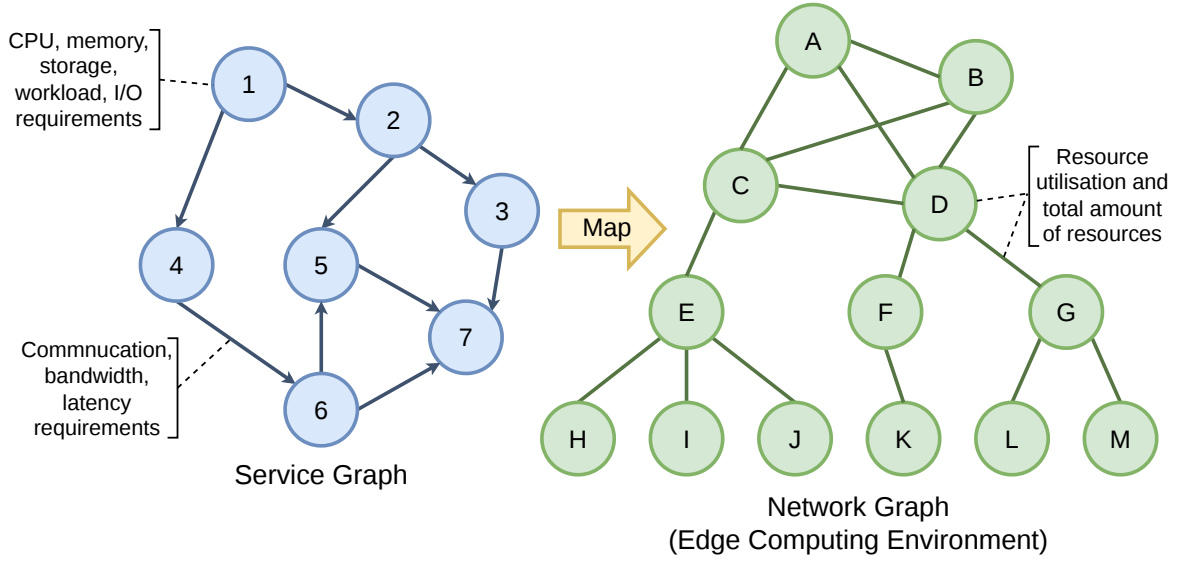


Figure 2.6: The service placement process: mapping the service graph to the network graph.

fibre links (e.g., dedicated inter-MDC backhaul connections) or via high-bandwidth wireless mesh channels. The physical topology shown is structured hierarchically, where the bottom tier resembles a tree structure. This tree topology models the hierarchical aggregation of the RAN, where multiple distributed access switches and base stations at the network periphery aggregate traffic upward into higher-tier edge servers and gateways before reaching the core network and central cloud.

Based on this mapping pattern, services are subsequently deployed onto specific computing devices by the platform manager. Mathematically, the service placement problem is formulated as finding a decision matrix that maps V application modules to M physical servers while mapping inter-module data dependencies to valid physical routing paths. The placement process must search the vast combinatorial solution space of the physical network for available computational and communication resources that satisfy all functional requirements and strictly adhere to operational constraints, while simultaneously co-optimising key performance indicators such as latency and energy consumption [133]. A detailed description of the most critical constraints is provided in Section 2.4.2.

2.4.2 Constraints and Challenges

In the service placement problem, system architects must address distinct categories of requirements. These are summarised as follows:

Computing-oriented: Service modules require a specific amount of CPU power, number of cores, memory, and storage to be deployed and executed successfully [44].

Network-oriented: Services may impose strict bandwidth or network latency requirements on communication paths, including internal connections between microservices and external links to end devices. Constraints may also limit the acceptable packet error rate and link jitter [157] to support reliable data transmission.

Task-oriented: Applications often enforce strict deadlines for task completion times, typically defined within SLAs [121]. For continuously running stream processing services, requirements may focus on processing throughput [15], specifying a minimum acceptable value rather than a single completion deadline.

Location-oriented: In specific scenarios, an application or service module is geographically pinned; that is, it is constrained to run only at specific edge servers due to data sovereignty laws, security policies, or specific hardware dependencies (e.g., a local GPU or sensor interface) [107].

Satisfying these diverse constraints requires navigating the complex trade-offs inherent in distributed systems [46]. For example, placing a service on the geographically closest edge server fulfils network constraints by minimising propagation delay; however, it may cause server congestion if the server lacks available resources. Conversely, offloading the task to a distant but more powerful edge server meets computing constraints by providing adequate processing power, but it introduces higher communication latency that risks violating strict task deadlines.

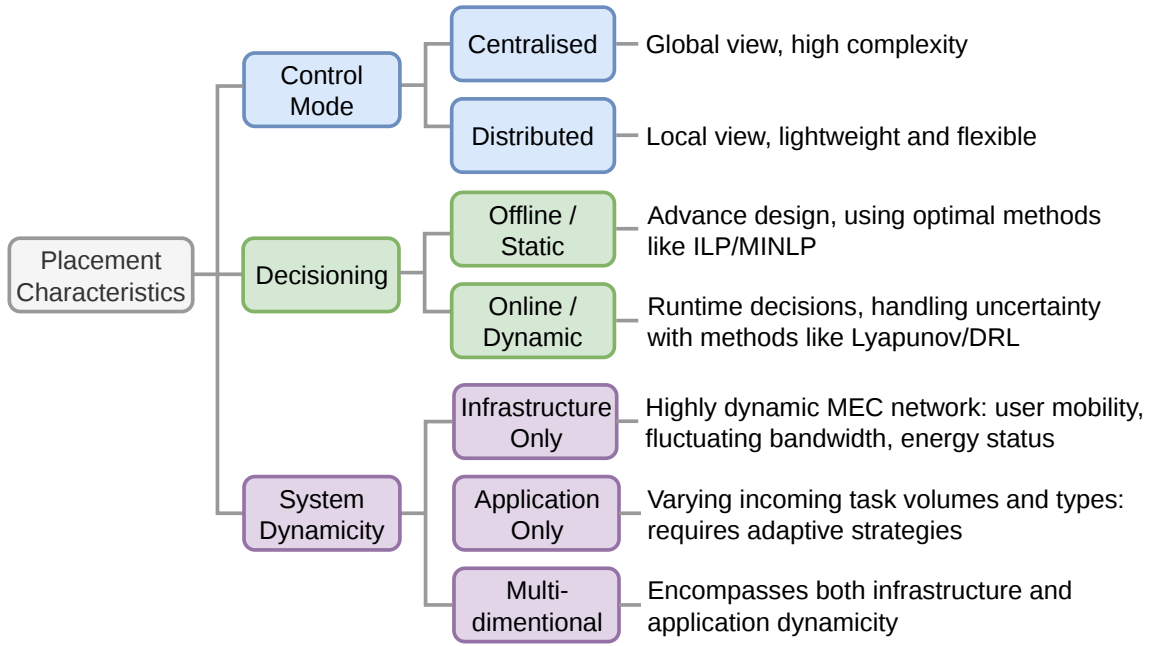


Figure 2.7: The taxonomy tree of service placement characteristics.

2.4.3 Placement Characteristics

Before designing deployment strategies, the key characteristics of the specific placement scenario must be carefully considered. Here, we summarise the following characteristics to classify service placement problems found in the literature. Figure 2.7 illustrates the proposed taxonomy of these service placement characteristics.

Control Mode: The control mode adopted by the system for coordination can be either centralised or distributed. For systems with multi-layered topologies composed of geo-distributed entities, these two modes present distinct advantages and trade-offs.

- **Centralised:** Under the centralised control mode, all coordination events are orchestrated from a central control plane. Global information regarding all infrastructure resources and application demands is required to make global deployment decisions. Gathering this aggregate information increases the potential for finding a globally optimal solution to the placement problem

[101]; however, deriving the absolute optimal placement incurs prohibitive computational overhead at the control plane.

- **Distributed:** In contrast, a distributed approach utilises multiple orchestrator modules dispersed throughout the network to control service placement. Each management module collects local resource and requirement information to compute placement decisions separately [97]. This control mode is flexible and lightweight, making it more efficient when handling distributed computing resources across wide network ranges. While this mode improves the experience of local users by reducing control-plane latency, its decisions may lack global optimality due to the absence of a holistic system view.

Decisioning: The approach to service placement can be categorised as either offline (static) or online (dynamic).

- **Offline Placement:** Decisions are made in advance during the design or compile stage, where all required information is available. Complete knowledge of system specifications and placement demands is needed to provide solutions. Many offline placement algorithms proposed in the literature utilise exact methods, such as Integer Linear Programming (ILP) [159] or MINLP [66], to find optimal solutions for static scenarios.
- **Online Placement:** Conversely, online placement decisions are made during system runtime. In this case, the system utilises both initial information and the current, evolving system state. For real-world cases like vehicular networks or crowdsourced live streaming, where services continuously arrive or edge entities may be disconnected or newly connected [140], placement decisions cannot be pre-calculated. An online placement strategy can better accommodate time-varying changes; however, because it lacks knowledge of future events, it is challenging to achieve the global optimum of system resources. For online placement problems, strategies leveraging Lyapunov optimisation [141] and Deep Reinforcement Learning (DRL) [73] have been proposed to handle runtime uncertainties.

System Dynamicity: System dynamicity determines whether the placement strategy must address dynamic changes within the environment. This dynamicity manifests on the infrastructure side, the application side, or concurrently across both.

- **Infrastructure:** The physical and network infrastructure in MEC environments is inherently volatile. Mobile user devices frequently transition across coverage zones or drop offline, triggering unpredictable spatial and temporal shifts in communication demands and network congestion [124]. Consequently, wireless link capacities, such as available bandwidth and latency, fluctuate constantly due to channel fading and interference. Furthermore, the operational capacity of the infrastructure itself is not deterministic; in energy-harvesting edge deployments, the available computing resources and battery levels of edge servers are subject to environmental conditions and change continuously over time [153].
- **Applications:** At the application layer, user compute demands exhibit high degrees of heterogeneity and burstiness. The volume, type, and complexity of incoming tasks can spike unpredictably in response to real-world events or user behaviours, encompassing everything from lightweight data queries to intensive, interdependent service workloads [41]. Because these service requirements and resource footprints are rarely constant, static placement models inevitably lead to resource bottlenecks or QoS degradation. Instead, the placement problem dictates highly dynamic, adaptive strategies capable of continuously monitoring fluctuating workloads to deploy, migrate, scale, or tear down services on the fly [140].

2.5 Summary

This chapter has established the foundational architecture and operational paradigms of MEC. By extending cloud-like capabilities to the network periphery, MEC is uniquely positioned to satisfy the ultra-low latency and high-bandwidth requirements of modern, data-intensive applications. To fully

leverage this distributed infrastructure, contemporary edge workloads are increasingly modelled as interconnected services, often represented as DAGs. While this structural paradigm offers significant application flexibility, it simultaneously introduces intricate service dependencies that complicate execution and resource allocation.

The fundamental conflict within this ecosystem arises from the tension between these sophisticated application demands and the physical constraints of the edge infrastructure. MEC hosts, particularly those deployed in remote or off-grid locations, must operate within strict computational limits and highly variable energy budgets. This challenge is further compounded when relying on intermittent renewable energy sources via energy harvesting techniques. Consequently, sustaining complex service dependencies without violating strict QoS constraints or exhausting local power reserves remains a critical open issue.

Resolving these competing demands requires moving beyond static deployments towards dynamic, intelligent resource orchestration. The necessity for advanced service placement strategies forms the crux of the following chapter. Chapter 3 will present a comprehensive review of the state-of-the-art literature addressing these placement challenges.

Chapter 3

Literature Review

This chapter provides a comprehensive review of the state-of-the-art literature surrounding resource allocation and service placement in MEC environments, establishing the context for the methodological contributions of this dissertation.

Section 3.1 explores the foundational aspects of service placement optimisation, detailing the primary performance objectives alongside the diverse methodological approaches employed by the literature. Section 3.2 narrows the focus to dependency-aware service placement. It reviews how current literature models inter-service dependencies and critically examines existing strategies for co-optimising latency and energy for complex tasks.

Section 3.3 investigates the intersection of MEC and green computing, reviewing online scheduling, cloud offloading, and power management techniques in Energy-Harvesting enabled edge systems. It contrasts approaches for hybrid-powered systems against the strict operational limitations of purely renewable, off-grid deployments. Section 3.4 evaluates the standard simulation environments, tools, and testbeds utilised by researchers to validate the placement and scheduling algorithms.

Finally, Section 3.5 provides a critical assessment of the reviewed literature, synthesising the major

unresolved methodological gaps, particularly the high computational overhead of existing server-side energy optimisation strategies for dependent tasks and the absence of depletion-safe scheduling mechanisms, and Section 3.6 explicitly maps these gaps to the novel contributions presented in the subsequent chapters of this work.

3.1 Service Placement Optimisation

3.1.1 Optimisation Objectives

In the context of edge computing, the efficacy of any service placement or resource management strategy is evaluated against specific optimisation objectives. Because edge environments are inherently resource-constrained and application demands vary widely, these objectives often present complex trade-offs, such as balancing rapid execution against power limitations or deployment expenses. The following outlines the primary performance, cost, and sustainability metrics most frequently targeted in the literature to guide algorithmic design and evaluate system efficiency.

End-to-end Latency: This metric represents the total time experienced by a user from the moment a request is sent to the moment the result is received. It encompasses the time required for a request to travel from source to destination and be processed, making it highly dependent on transmission media, link propagation, routing, and storage delays. For delay-sensitive applications, achieving low service latency is paramount. Consequently, minimising end-to-end latency has attracted significant attention, with researchers heavily focused on architectural placement to reduce delay. For instance, Harutyunyan et al. [49] solves the joint user association and SFC placement problem in 5G networks to minimise latency, while Centofanti et al. [16] introduces an architecture for placing services onto cluster nodes specifically designed to offer lower end-to-end latency.

Energy Consumption: As a primary concern in edge systems, energy consumption remains a critical performance metric. Total energy consumption depends not merely on the number and types

of servers deployed or their internal resource utilisation, but also on the communication overhead between network entities. To reduce energy consumption and increase efficiency, the energy management techniques introduced in Section 2.3.2 are frequently adopted. Literature typically addresses this challenge from the perspective of the UE side, the server side, or both. Minimising energy consumption at the device level extends battery life, while optimising server-side energy is critical for system sustainability, particularly in energy-constrained environments. For example, PremSankar and Ghaddar [108] proposes heuristics for placing artificial intelligence applications to reduce edge energy consumption. Other studies aim to optimise the energy consumption of the entire system. Work in [155] designs a service migration method to minimise total energy consumption across both the vehicles and edge server sides for the Internet of Vehicles. Similarly, a partial computation offloading method is proposed in [12] to minimise the total energy consumed by both mobile devices and edge servers. Furthermore, assessing the environmental footprint of edge systems has become increasingly important. Because CO₂ emissions are directly tied to both the volume of energy consumed and the nature of the power source, they serve as a vital sustainability metric. In this context, Ahvar et al. [2] proposes a dynamic application placement and migration strategy designed to simultaneously minimise energy expenditures and carbon emissions in edge clouds.

Operational Cost: Cost-related factors play a pivotal role in platform management and can represent any financial overhead associated with the placement problem. Typical operational costs include networking overhead, server leasing expenses, and grid electricity usage, which may be billed through time-based pricing, task-granularity workloads, or bandwidth consumption. These costs heavily influence decision-making from the perspectives of both service providers and end-users. For example, Azizi et al. [7] treats latency and the deployment costs of service replicas across fog and cloud resources as primary optimisation objectives. Faraji-Mehmandar et al. [40] formulates the execution cost of service requests as a composite of computational and communication overheads. Similarly, Rac and Brorsson [111] introduces a cost-effective scheduling methodology that streamlines dis-

tributed application deployment while actively reducing CPU and network usage expenses.

Resource Utilisation: Achieving optimal resource utilisation at the network edge remains a fundamental challenge. A core trade-off exists in service placement: while consolidating services onto a single edge server minimises inter-service communication latency, it risks bottlenecking the node and degrading the overall user experience. Conversely, distributing services across disparate nodes balances the computational load but incurs significant network communication overhead. Strategic placement is therefore imperative to mitigate resource fragmentation and maximise the limited capacity of edge servers. Addressing this, Lin et al. [70] proposes a service scheduling algorithm that balances resource utilisation against dynamic request volumes. Skarlat et al. [130] investigates strategies for improving general resource utilisation across the broader edge infrastructure, and for fog environments, Wahab et al. [144] jointly optimises QoS and hardware utilisation.

Throughput: Task processing throughput is a critical performance indicator, particularly for data-intensive applications such as stream processing. Typically measured by the number of successful requests processed per unit time or the volume of data transmitted per second, this metric reflects the system's overarching computational capacity and its ability to sustain high-traffic workloads. Lin et al. [71] places IoT services on fog nodes to seek an optimal trade-off among service costs, response latency, resource utilisation, and system throughput. In multi-cell MEC architectures, Liang et al. [69] introduces a resource scheduling mechanism designed to maximise MEC throughput while simultaneously minimising migration overhead. For pervasive edge computing networks, Ning et al. [99] formulates a dynamic service placement optimisation problem aimed at maximising overall system utility.

Other Metrics: Beyond standard performance indicators, metrics such as model accuracy and system reliability frequently drive optimisation efforts. For example, Hudson et al. [56] optimises service delay alongside artificial intelligence model accuracy in edge intelligence scenarios. Maintaining system stability in highly dynamic environments is another crucial parameter; Zhang et al. [160] manages

Metric	Primary Focus & Description	Typical Units / Formulation
End-to-end Latency	Total delay from request initiation to response receipt, encompassing transmission, queueing, and processing times.	Seconds (s) / ms; $T_{\text{total}} = T_{\text{comm}} + T_{\text{comp}}$
Energy Consumption	Electrical energy consumed during task execution. Categorised into UE battery drain vs. Server dynamic CPU power.	Joules (J) / kWh; $E = P \times T_{\text{exec}}$
Operational Cost	Financial expenditures incurred by network providers or users, encompassing server leasing and grid power prices.	Monetary units (\$, £); $\sum C_{\text{resource}}$
Resource Utilisation	Ratio of active computational load (CPU cycles, memory allocation) against total available MDC infrastructure capacity.	Percentage (%); $\frac{\text{Used Capacity}}{\text{Total Capacity}}$
Throughput	Rate of successful task completions or data volume processed over a given operating window.	Requests/sec or Gbps

Table 3.1: Summary of primary optimisation metrics and their mathematical definitions in the literature.

task offloading while maintaining strict queue stability within Industrial IoT scenarios. Furthermore, the failure rate, as a metric reflecting the frequency of unsuccessful task executions or dropped requests due to node mobility, resource exhaustion, or network interruptions, is highly prioritised. In Reference [138], a service placement and user assignment model is proposed to minimise the worst-case penalty and improve the overall ratio of successfully served requests.

To synthesise these diverse evaluation criteria, Table 3.1 summarises the primary optimisation metrics, detailing their core focus and typical mathematical formulations across the surveyed literature.

3.1.2 Methodological Approaches

The service placement problem is a notoriously challenging combinatorial optimisation problem, as the number of possible solutions grows exponentially with the size of the network. Computing optimal application scheduling and service placement within edge infrastructure is inherently an NP-hard problem [31, 139], with domain-specific constraints further exacerbating its complexity. To address

these complex single- and multi-objective optimisation problems, the literature explores a diverse array of methodological approaches, ranging from traditional exact methods to adaptive, learning-based techniques.

Exact Methods: Many researchers have proposed exact approaches based on mathematical models, most notably ILP and MILP. In ILP/MILP formulations, the optimisation problem is represented by a linear objective function subject to discrete or continuous constraints. To find the optimal placement solution, researchers typically perform exhaustive searches or employ commercial solvers such as IBM ILOG CPLEX and Gurobi. For example, to minimise the total delay of all services, the authors in [86] formulate the service placement problem for edge-cloud infrastructures as a MILP problem and solve it using CPLEX. Similarly, Taghavian et al. [137] proposes a deterministic service placement solution structured around a Branch and Bound algorithm, which allows for a complete search over the feasible solution space. However, due to the NP-hard nature of the problem, these exact methods are typically limited to small-scale scenarios or utilised strictly as offline benchmarks. The execution of such algorithms is computationally intensive and time-consuming, rendering them unsuitable for large-scale environments or latency-sensitive use cases where real-time decision-making is required.

Heuristics: In complex mobile and edge infrastructures, exact analysis can be practically inapplicable due to the vast solution space. To overcome the computational complexity of exact methods and reduce the massive search space of this combinatorial problem, heuristic algorithms are widely adopted. Heuristics consist of a specific set of rules and logic tailored to the problem, designed to find near-optimal candidate solutions much faster than exhaustive methods. For example, PremSankar and Ghaddar [108] proposes a lightweight heuristic for placing artificial intelligence applications based on greedy policies, while Hassan et al. [50] introduces a priority-, network-, and energy-aware heuristic for IoT application placement in fog-cloud systems. However, such heuristic-based approaches generally lack theoretical performance bounds and are susceptible to getting trapped in local optima.

Meta-heuristics: To enable a more in-depth exploration of promising areas, navigate larger solu-

tion spaces, and avoid the pitfalls of local optima, meta-heuristic algorithms are frequently utilised in the literature. Population-based meta-heuristics often mimic evolutionary selection or the swarm behaviour of animals. Algorithms such as Genetic Algorithms (GA), Ant Colony Optimisation, and Particle Swarm Optimisation are highly prevalent. These solutions iteratively attempt to improve result quality by searching for better placement options through stochastic and evolutionary mechanisms. In Reference [167], a genetic algorithm based on simulated annealing is proposed to optimise service placement in MEC environments, aiming to balance user QoS with edge server energy consumption. Huang et al. [55] proposes an Ant Colony Optimisation algorithm to place service instances for IoT applications in the fog, minimising both deployment costs and service latency. Furthermore, a Quantum-Inspired Particle Swarm Optimisation algorithm is introduced by Bey et al. [9] to address IoT service placement while simultaneously optimising the throughput, energy consumption, delay, and computational load of the system. However, despite their enhanced global search capabilities, these population-based approaches generally incur significant computational overhead and slow convergence rates, making them challenging to deploy in highly dynamic, ultra-low-latency environments that demand strict real-time decision-making.

Control theory: Control-theoretic approaches (such as Lyapunov optimisation) are frequently employed to manage highly dynamic environments by establishing mathematical bounds for system stability, such as bounding queue backlogs without requiring prior knowledge of network statistics. For example, Tang et al. [141] proposed a service placement algorithm based on Lyapunov optimisation in vehicular edge computing, while Ma et al. [82] designs a service placement algorithm using Lyapunov optimisation to predict user mobility in the near future. However, these techniques typically focus on long-term time-averaged performance, which can lead to suboptimal transient behaviour, and they inherently introduce a mathematical trade-off between system delay (e.g., queue length) and the primary optimisation objective (e.g., energy consumption).

Learning-based Driven by the highly dynamic and stochastic nature of the Internet of Everything and

the continuous influx of big data, researchers are increasingly turning to learning-based and control-theoretic approaches. Machine learning models, which extract patterns from real execution data, are widely used to proactively generate service placement solutions. For highly dynamic scenarios with multiple fluctuating factors (e.g., user mobility, renewable energy generation), these approaches excel at monitoring system status and making adaptive decisions. The learning-based methods used in the literature can generally be categorised into three groups: Supervised Learning, Reinforcement Learning, and Deep Learning. Supervised Learning involves training and evaluating models on labelled datasets composed of known inputs and optimal outputs; for example, Manikandan et al. [88] proposes a decision tree to schedule IoT-based smart healthcare applications. Reinforcement Learning models learn within an interactive environment through trial-and-error actions and delayed rewards; for instance, Flegkas et al. [42] proposes a Q-learning-based algorithm to determine optimal service locations across the 6G edge-to-cloud continuum. Deep Learning utilises high-dimensional neural networks to model complex, non-linear relationships within input data; Liu and Zhang [74] present a DRL framework for SFC embedding in large-scale edge-cloud networks. However, learning-based approaches suffer from notable drawbacks, including prolonged training times, high computational overhead during the inference phase, challenges in achieving convergence stability, and a distinct lack of interpretability, which complicates their deployment in mission-critical systems requiring strict safety or reliability performance.

3.2 State-of-the-Art in Dependency-Aware Service Placement

To comprehensively review the state-of-the-art in dependency-aware service placement, the following discussion categorises the existing literature across three structural dimensions: formal problem formulations and system models, algorithmic placement strategies, and an overarching comparative synthesis highlighting key methodological gaps.

Section 3.2.1 establishes the mathematical foundation by reviewing how applications are topologi-

cally represented alongside the formal latency and energy equations defined across studied scenarios. Section 3.2.2 categorises placement strategies based on their primary optimisation objectives, contrasting latency-centric methods against multi-objective co-optimisations and domain-specific edge architectures. Finally, Section 3.2.3 provides a comprehensive comparison matrix in Table 3.2 and synthesises the critical gap regarding server-side energy efficiency for multi-tenant dependent workloads.

3.2.1 Formal Problem Formulations and System Models

Service placement for dependent tasks differs fundamentally from independent task offloading. Rather than making isolated mapping decisions for individual requests, placement algorithms must map structured application topologies onto physical network graphs while co-optimising spatial server allocation and temporal task execution order.

Application Topologies and Workload Representation

In the literature, dependent applications are represented using three primary graph abstractions:

- **DAGs:** The predominant model represents an application as a DAG $G = (V, E)$, where vertices $v_i \in V$ denote subtasks with computational workloads c_i (measured in CPU cycles), and directed edges $(v_i, v_j) \in E$ signify data dependencies requiring $D_{i,j}$ bits of transmission. A directed edge enforces a strict precedence constraint: subtask v_j cannot begin execution until its predecessor $v_i \in \text{Pred}(v_j)$ finishes execution and all output data $D_{i,j}$ is transferred. The graph contains a set of entry tasks V_{entry} (no predecessors) and exit tasks V_{exit} (no successors). Studies by Chen et al. [23], Shang et al. [126], Shen et al. [127], Xu et al. [152], and Song et al. [132] rely on this general DAG model.
- **SFCs and Microservice Chains:** In VNF and microservice architectures, dependencies follow an ordered chain $SC = \langle ms_1, ms_2, \dots, ms_k \rangle$ or a sequential workflow. In these models, data flows

sequentially through microservice instances. Recent works by Hu et al. [53] and Wang et al. [146] extend chain models to multi-instance container deployments, where microservices are deployed across multiple edge nodes and shared among parallel user requests. In satellite edge computing, Xia et al. [150] explicitly incorporates VNF state data migration size $\alpha_k b_k$ between adjacent virtual functions.

- **Stream Processing Graphs:** For real-time streaming applications, Khare et al. [61] models applications as continuous stream DAGs where nodes represent stream operators and edges denote continuous data stream flows. Unlike discrete batch tasks, operators run continuously, making co-location resource interference (CPU cache contention, memory bandwidth sharing) the dominant performance bottleneck.

Mathematical Formulations of Latency Models

Depending on the application topology and network architecture, end-to-end latency is formulated mathematically in the literature through distinct mechanisms:

DAG Completion Time (Makespan): For DAG applications, overall latency is defined by the finish time of the critical path (the longest sequence of dependent computation and communication delays from entry to exit tasks). For a subtask v_i assigned to computing node m , its computational processing delay $T_{\text{comp}}(v_i, m)$ is calculated using the following equation:

$$T_{\text{comp}}(v_i, m) = \frac{c_i}{f_m} \quad (3.1)$$

where f_m is the assigned CPU operating frequency of node m . The inter-task communication latency $T_{\text{comm}}(v_i, v_j)$ between predecessor v_i (placed on node m) and successor v_j (placed on node n) is

expressed as follows:

$$T_{\text{comm}}(v_i, v_j) = \begin{cases} 0 & \text{if } m = n \\ \frac{D_{i,j}}{R_{m,n}} & \text{if } m \neq n \end{cases} \quad (3.2)$$

where $R_{m,n}$ represents the data transmission rate of the network link between nodes m and n . Consequently, the Earliest Start Time $\text{EST}(v_i)$ and Earliest Finish Time $\text{EFT}(v_i)$ of subtask v_i are constrained by precedence waiting delays:

$$\text{EST}(v_i) = \max \left\{ a_i, \max_{v_k \in \text{Pred}(v_i)} (\text{EFT}(v_k) + T_{\text{comm}}(v_k, v_i)) \right\} \quad (3.3)$$

$$\text{EFT}(v_i) = \text{EST}(v_i) + T_{\text{comp}}(v_i) \quad (3.4)$$

where a_i is the task arrival time. The overall end-to-end application latency (makespan) is the maximum finish time across all exit tasks:

$$T_{\text{makespan}} = \max_{v_e \in V_{\text{exit}}} \{\text{EFT}(v_e)\} \quad (3.5)$$

This formulations is adopted by Chen et al. [23], Shang et al. [126], Zhang et al. [163], and Song et al. [132].

Queuing Network Latency: In multi-instance microservice environments, Hu et al. [53] and Wang et al. [146] model edge nodes as M/M/C queuing networks. End-to-end latency is formulated as the cumulative sum of request buffering delays, CPU execution times, and network propagation delays across service instances:

$$T_{\text{E2E}} = \sum_{m=1}^K \left(T_{\text{queue}}^{(m)} + T_{\text{exec}}^{(m)}(f_m, U_m) + T_{\text{trans}}^{(m,m+1)} \right) \quad (3.6)$$

where queueing delay $T_{\text{queue}}^{(m)}$ depends on arrival rates λ_m and active container capacities, while execu-

tion delay $T_{\text{exec}}^{(m)}$ scales dynamically with CPU frequency f_m and utilisation load U_m .

Specialised Delay Metrics: In Satellite Edge Computing, Xia et al. [150] incorporates satellite-to-ground coverage visibility time slots Δt_{vis} and VNF state transfer delay $T_{\text{state}} = \alpha_k b_k / R_{\text{sat}}$, while Khare et al. [61] incorporates a co-location interference penalty β_{interf} into execution latency.

Mathematical Formulations of Energy Consumption Models

A key differentiator across the surveyed literature lies in how energy consumption is modelled and whose energy budget is prioritised:

Mobile Device / UE Energy: In user-centric studies [23, 126, 89, 127, 152], total energy consumption evaluates the battery drain of mobile devices. For local CPU processing, dynamic power scales quadratically with frequency via DVFS, while wireless transmission consumes power over the uplink duration:

$$E_{\text{UE}} = \sum_{v_i \in V_{\text{local}}} \kappa \cdot f_{\text{local}}^2 \cdot c_i + \sum_{v_j \in V_{\text{offload}}} P_{\text{tx}} \cdot \left(\frac{D_j}{R_{\text{uplink}}} \right) \quad (3.7)$$

where κ is the effective switched capacitance of the mobile CPU hardware, and P_{tx} is the RF transmit power. Remote execution energy consumed by edge or cloud servers is explicitly excluded from the optimisation budget.

Server-Side Dynamic and Idle Energy: In infrastructure-centric studies [53, 146], edge server power dissipation is modelled as the sum of baseline static standby power P_{static} and dynamic CPU power P_{dynamic} , which scales cubically with operating frequency f_m and linearly with resource utilisation U_m :

$$P_{\text{server}}(m) = P_{\text{static}} + \gamma \cdot f_m^3 \cdot U_m \quad (3.8)$$

Total server energy E_{server} integrates power over operating time slots Δt : $E_{\text{server}} = \sum_m \int P_{\text{server}}(m) dt$.

Multi-Tier and Hybrid Energy Models: In collaborative Unmanned Aerial Vehicle (UAV) networks

[98], total system energy incorporates mobile transmission energy, UAV computational energy, and aerodynamic UAV propulsion power $P_{\text{hover}} + P_{\text{flight}}$. In multi-tier green edge systems [81], server energy is split between harvested green energy battery queues (zero marginal cost) and grid power draw (monetary cost coefficient β).

3.2.2 Taxonomy and Analysis of Existing Placement Strategies

Based on the underlying optimisation objectives and structural domain constraints, dependency-aware placement strategies in the literature are categorised into three main paradigms:

Latency-Centric Placement Strategies

For latency-critical applications (such as autonomous driving perception or AR/VR rendering), algorithms prioritise minimising end-to-end latency without constraints on energy consumption. Khare et al. [61] proposes the Linearize-Predict-and-Place heuristic for stream processing DAGs, converting arbitrary graphs into linear chains, predicting operator co-location interference via regression, and placing operators to minimise makespan. Chen et al. [23] addresses completion time minimisation for mobile DAG offloading, using heterogeneous-earliest-finish-time-based task ordering combined with convex optimisation for edge-cloud bandwidth allocation. Mo et al. [91] extracts function-level call graphs and applies dynamic programming along critical paths to identify optimal offloading boundaries under fixed user energy budgets. Yu et al. [158] models MEC-cloud partial offloading as a tandem queue and applies DRL to dynamically adjust compute speeds to satisfy latency requirements. Xia et al. [150] investigates VNF placement in Satellite Edge Computing, constructing auxiliary transformation graphs to solve service chaining under time-varying low earth orbit satellite visibility constraints.

Energy-Aware and Joint Multi-Objective Co-Optimisation

Because aggressive offloading to achieve minimal delay can rapidly deplete battery reserves or overload edge infrastructure, the vast majority of recent literature focuses on joint latency-energy co-optimisation, typically minimising a weighted objective function $J = \alpha T_{\text{makespan}} + (1 - \alpha) E_{\text{total}}$.

User-Centric Energy Minimisation (Dominant Paradigm): The prevailing trend in the literature focuses on maximising mobile battery longevity. Shang et al. [126] proposes a heuristic decomposing user energy budgets across DAG subtasks while enforcing deadline T_{max} . Mehrabi et al. [89] formulates an MINLP model for sequential task offloading with edge service caching, deploying a greedy heuristic to minimise mobile battery drain. Shen et al. [127] addresses task offloading and service caching in Vehicle Edge Computing (VEC), applying DRL and cyclic iterative scheduling to reduce vehicle battery depletion and processing delay. Zhang et al. [163] uses game theory and graph partitioning to strike a trade-off between mobile energy consumption and DAG completion delay. Xu et al. [152] applies an improved genetic algorithm to schedule industrial IoT DAG tasks, minimising terminal device energy consumption in smart factories. Song et al. [132] introduces a Transformer-integrated Proximal Policy Optimisation framework to encode complex DAG topologies, co-optimising transmission power and CPU frequency to reduce application makespan and user energy.

Joint User and System Energy Minimisation: A smaller group of studies aggregates user transmission energy and server compute energy into a single joint cost. Chen et al. [19] uses an Actor-Critic DRL mechanism combined with heterogeneous-earliest-finish-time priority scheduling to optimise multi-user DAG offloading, evaluating total aggregated energy across mobile devices and shared edge servers. Nguyen et al. [98] deploys a hybrid meta-heuristic for UAV-assisted MEC, co-optimising user device battery drain, UAV hover propulsion energy, and task completion latency.

Server-Side Infrastructure Energy Minimisation: Shifting focus away from mobile batteries, re-

cent microservice studies investigate server-side power state management. Hu et al. [53] proposes the Global-Reward Soft Actor-Critic algorithm for microservice chain orchestration, dynamically adjusting server DVFS operating frequencies and sleep states to reduce edge server power dissipation while meeting SLA deadlines. Wang et al. [146] presents Delay-Aware Reward Scaling Proximal Policy Optimisation for multi-instance microservice deployment, joint-optimising container routing and DVFS scaling to minimise total edge server dynamic energy.

Specialised Domain-Specific Architectural Variations

Placement strategies are further shaped by domain-specific physical environments:

- **VEC:** Shen et al. [127] incorporates high vehicle mobility, short contact times, and Roadside Unit service caching hit/miss overheads into DAG placement.
- **Drone-Assisted MEC (UAV):** Nguyen et al. [98] models dynamic Line-of-Sight air-to-ground wireless channels and joint UAV trajectory planning alongside task dependency offloading.
- **Satellite Edge Computing:** Xia et al. [150] addresses dynamic orbit topologies, high inter-satellite propagation delay, and VNF state data migration overhead.
- **Fog Computing & Reliability:** Farhadi et al. [41] and Pallewatta et al. [102] address microservice placement across fault-prone fog nodes, formulating ILP models to maximise application execution reliability (SLA compliance under node failures) while managing shared storage datasets.

3.2.3 Literature Synthesis and Methodological Gaps

To provide a comprehensive comparative synthesis of the state-of-the-art in dependency-aware service placement, Table 3.2 systematically categorises 16 representative studies alongside the contribution of this dissertation across target scenarios, dependency models, optimisation objectives, energy focus,

latency scope, energy scope, problem formulations, and solution methodologies.

In Table 3.2, aside from the aforementioned domain-specific architectures, we distinguish between 'MEC' and generic 'Edge' computing scenarios based on their architectural boundaries: studies classified under 'MEC' explicitly model ETSI-standardised MEC architectures integrated directly within cellular RAN, whereas studies classified as 'Edge' adopt a generalised edge computing abstraction (such as local Wi-Fi servers or IoT gateways without specific cellular network constraints).

The Methodological Gap

A critical analysis of Table 3.2 reveals a distinct methodological trade-off in existing literature. While extensive research addresses dependency-aware service placement, the vast majority of studies focus on preserving mobile device battery life (E_{UE}), treating edge servers as unconstrained execution sinks with static power profiles. Conversely, modern literature addressing server-side energy [53, 146] predominantly adopts learning-based approaches. Although capable of handling complex state spaces, these paradigms require extensive offline training time and high computational overhead, limiting their suitability for rapid, real-time edge adaptation. Furthermore, these works frequently simplify application topologies to linear service chains or uncoordinated queuing models, failing to capture the complex inter-task precedence constraints of general DAGs.

Managing server-side energy dissipation for complex DAG workloads in resource-constrained edge MDCs requires balancing execution latency against non-linear server power dynamics. Existing exact methods suffer from computational intractability, whereas learning-based schemes demand long training phases. Resolving these operational limitations motivates the rapid, low-overhead heuristic methodology detailed in Chapter 4.

Reference	Target Scenario	Dependency Model	Optimisation	Energy Focus	Latency Model Scope	Energy Model Scope	Problem Formulation	Methodology
[23]	MEC	Tree-Graph	Latency	User	DAG Makespan (exit task finish time)	Mobile battery ($E_{\text{local}} + E_{\text{tx}}$)	Joint subtask offloading & resource allocation under energy budget	Heuristic
[19]	Edge	DAG	Latency & Energy	Both	Multi-user DAG Makespan	Mobile battery + edge server dynamic energy	Multi-user multitask offloading under latency & server capacity	DRL
[53]	Edge	Sequential	Latency & Energy	Server	M/M/C Queuing & Execution delay	Server power ($W_{\text{static}} + W_{\text{dynamic}}(f, U)$) via DVFS	Microservice deployment & server power state control	DRL
[61]	Edge	DAG	Latency	N/A	Stream Makespan & co-location interference	N/A (Resource capacity bounds)	Stream operator placement managing co-location interference	Heuristic
[89]	MEC	Sequential	Latency & Energy	User	Task completion delay ($T_{\text{local}} + T_{\text{tx}} + T_{\text{edge}}$)	Mobile battery ($E_{\text{local}} + E_{\text{tx}}$)	Joint offloading, service caching & resource allocation	Exact Method
[91]	MEC	DAG	Latency	N/A	Critical path latency	User battery budget ($E \leq E_{\text{max}}$)	Function-level task offloading under user energy budget	Heuristic
[98]	VEC	DAG	Latency & Energy	User	Air-to-ground & inter-UAV relay delay	Mobile battery + UAV flight & compute power	Joint offloading, UAV trajectory & bandwidth allocation	Meta-heuristic
[102]	Fog Computing	DAG	Latency & Reliability	Server	Microservice queuing & network delay	Fog node & gateway infrastructure energy	Proactive microservice placement maximising reliability	Exact Method
[126]	MEC	DAG	Latency & Energy	User	DAG Makespan + Deadline T_{max}	Mobile battery ($E_{\text{comp}} + E_{\text{trans}}$)	Maximise reliability under energy budget & deadline	Heuristic
[127]	VEC	DAG	Latency & Energy	User	transmission + execution + cache miss	Vehicle battery energy depletion	Joint offloading & service caching	Heuristic
[132]	Edge	DAG	Latency & Energy	User	Heterogeneous DAG Makespan	System energy (Edge compute f^3 + link tx)	DAG AI app scheduling in dynamic edge	DRL
[146]	Edge	Sequential	Latency & Energy	Server	M/M/C Queuing & propagation delay	Server standby + dynamic CPU power $\propto f^3 U$	Microservice deployment & DVFS frequency control	DRL
[150]	Satellite Edge	Sequential	Latency	N/A	Satellite visibility + transmission + state transfer	N/A (Satellite capacity bounds)	Joint VNF placement & routing under low earth orbit coverage	Heuristic
[152]	MEC	DAG	Latency	User	DAG total application delay	Industrial terminal battery	Industrial DAG task offloading in smart factory	Meta-heuristic
[158]	MEC	Sequential	Latency	N/A	Parallel execution delay $\max(T_{\text{local}}, T_{\text{offload}})$	Mobile battery budget ($E_{\text{local}} = \kappa f^3$)	Partial offloading & power/frequency allocation	DRL
[163]	MEC	DAG	Latency & Energy	User	DAG Makespan + dependency wait time	Mobile battery ($E_{\text{comp}} + E_{\text{trans}}$)	Dependent task offloading under energy budget	Heuristic
[25] (Ch. 4)	MEC	DAG	Latency & Energy	Server	DAG Makespan	Server dynamic ($f^3 U$)	Dependency-aware service placement prioritising server energy & latency	Heuristic (EDEM/DEM)

Table 3.2: Comparison of dependency-aware placement algorithms in the literature.

3.3 State-of-the-Art in Energy-Harvesting MEC Systems

3.3.1 Online Scheduling and Cloud Offloading

The Transition to Dynamic Environments

While Section 3.2 established that offline placement strategies are adequate for static scenarios, real world Edge Computing environments are inherently dynamic. Consequently, the literature increasingly emphasises the necessity of online scheduling to manage unpredictability stemming from two primary sources. First, user demand exhibits significant burstiness and time varying service requests. Studies by Li et al. [66] and Ma et al. [83] highlight how fluctuating mobile task arrivals require adaptive systems capable of decomposing long term optimisation problems into one shot decisions. Second, energy supply introduces a separate layer of stochasticity. Xia et al. [151] investigates the intermittent nature of renewable energy generation, noting that environmental energy harvesting creates a highly variable power supply that online systems must navigate effectively.

Managing Hybrid Energy and Operational Costs

To address energy unpredictability, a substantial portion of the literature focuses on systems equipped with both renewable energy capabilities and a reliable power grid connection. The primary objective across these studies is minimising the overall operational cost by determining the optimal moment to consume free but limited green energy versus purchasing reliable but expensive grid power. Chen et al. [28] and Chen et al. [27] propose distributed online resource allocation schemes that evaluate ongoing energy trading decisions with the smart grid alongside renewable energy availability. Similarly, Perin et al. [105] and Zhao et al. [165] develop predictive algorithms to reduce the burden on the traditional power grid by maximising the use of collocated renewable sources.

Methodologically, many of these approaches rely heavily on DRL or Lyapunov optimisation to balance operational costs with queue stability. Panda et al. [103] employs DRL to learn optimal data

distribution policies, while Zhao et al. [165] and Xia et al. [151] utilise perturbed Lyapunov optimisation to stabilise battery levels. However, these methods exhibit notable limitations in highly dynamic environments, such as slow convergence rates and high computational overhead.

Cloud Offloading as a Fallback Mechanism

When local edge servers experience computation overload or exhaust their green energy reserves, the standard solution in the literature is cloud offloading. This introduces a fundamental trade-off: processing workloads locally conserves network bandwidth but rapidly drains local energy, whereas offloading to the centralised cloud preserves edge energy but incurs transmission delays that risk violating SLAs. Li et al. [67] and Ma et al. [81] explore hierarchical architectures where remote cloud execution serves as a fallback, weighing local energy consumption against cloud rental fees and overall latency. Furthermore, Zhao et al. [166] and Kazemi et al. [59] utilise DVFS to reduce energy overhead, explicitly accepting increased latency as a necessary compromise to save power.

A crucial distinction among these studies is their treatment of strict deadlines. Some researchers treat service deadlines as hard constraints. For instance, Sriraghavendra et al. [135] utilises a genetic algorithm to dynamically determine placement, explicitly aiming to maintain response times within given deadlines. Conversely, other approaches merely apply a financial or structural penalty for late execution. Li et al. [68] minimises social cost by bounding computation workloads and assigning a specific expenditure penalty for dropped tasks, implying that deadline violations are permissible if the overall cost remains mathematically optimal. Additionally, Chen et al. [18] addresses deadlines by assigning sub-deadlines and utilising a reverse auction strategy, balancing execution cost against proactive scheduling.

The Methodological Gap

While existing online scheduling frameworks effectively manage independent workloads in hybrid energy-harvesting environments, extending these techniques to graph-structured dependent services introduces severe operational challenges. Conventional online offloading policies evaluate offloading decisions on an isolated per-task basis. However, in dependent microservice topologies, offloading a single subtask to a distant cloud without joint coordination of inter-service data dependencies causes severe communication latency cascades with co-located edge components. Currently, the literature lacks online scheduling mechanisms that adaptively balance local server scaling, dynamic cloud offloading, and inter-node transmission overhead under stochastic green energy generation and dynamic grid electricity pricing. Developing an integrated approach to co-optimize operational costs and QoS for complex service architectures under these dynamic constraints represents the primary focus and contribution of Chapter 5.

3.3.2 Power Management in Renewable Systems

The Off Grid Constraint

Unlike the hybrid setups discussed previously, systems relying solely on renewable energy face severe operational limitations. In purely energy harvesting MEC environments, energy is not merely an operational cost to be minimised but an absolute physical constraint. Hu et al. [52] and Xu et al. [154] highlight that devices powered exclusively by harvesting mechanisms require strict energy budgeting. To handle extreme mismatches between power supply and computational demand, traditional literature often resorts to aggressive admission control or extreme task deferment. For example, Ku et al. [64] examine solar powered roadside units, addressing solar intermittency through an offline energy scheduling phase combined with online user association to minimise service outages. Similarly, Aslanpour et al. [5] propose an energy aware scheduling algorithm for serverless edge systems, modelling resource allocation as an assignment problem to mitigate availability variance across nodes

powered by unpredictable renewable sources. Furthermore, researchers like Xu et al. [153] employ reinforcement learning for dynamic workload offloading to cope with the high intermittency of entirely off grid edge server provisioning. Zhou et al. [168] utilise cooperative game theory to allocate harvested energy and assign workloads to maximise throughput while respecting strict reliability constraints. Other approaches, such as the wake up scheduling concept by Rostami et al. [118], rely on duty cycling and sleep modes based on traffic prediction to reduce idle power consumption.

A prevalent technical lever utilised in the literature to stretch limited battery life is DVFS. State of the art methods actively leverage this technology to trade execution speed for lower dynamic power consumption. Wang et al. [148] demonstrate that jointly optimising computational speed via voltage scaling and offloading ratios significantly reduces energy consumption. Ki et al. [63] further combine frequency scaling with task offloading and hybrid memory allocation to save battery power in resource constrained environments. However, a critical pivot in this analysis is how these studies calculate the performance trade-off. The majority of these foundational works assume simple and independent tasks where slowing down the central processing unit frequency only delays one isolated output. While Chen et al. [22] advance the field by proposing a scheduling framework that exploits frequency downscaling while explicitly accounting for task dependencies within DAGs, the broader application of frequency scaling in entirely renewable systems remains largely focused on independent task models.

Treatments of Battery Safety

Beyond power reduction techniques, literature varies significantly in how it mathematically treats the battery state of charge. Many optimisation frameworks focus on maximising system utility or energy efficiency, but fewer promote depletion safety. Some researchers approach battery levels using a soft penalty function. For instance, Rakhmatov and Vrudhula [114] introduce a battery aware cost function where scheduling decisions that stress the battery incur soft penalties, though they also

formulate an endurance constraint requiring the battery to survive all scheduled tasks through a task repair procedure. Conversely, Chen et al. [21] enforce hard mathematical constraints for energy consumption in workflow scheduling to satisfy strict energy limitations. Another prominent approach involves Lyapunov drift plus penalty frameworks for queue and battery stability. Shi et al. [128] develop a Lyapunov optimisation strategy for battery energy storage systems, utilising a queue backlog mapped to discharged energy to enforce physical capacity limits. While these mathematically sound frameworks excel at maintaining long term averages, they frequently fail to avoid short term catastrophic battery depletion during sudden workload spikes or prolonged periods of limited renewable generation.

The Methodological Gap

Synthesising these threads reveals a critical methodological void in the existing literature. While extensive research leverages dynamic voltage scaling to save power, studies purely renewable setups, and attempts to manage battery levels, there is a profound lack of depletion-safe energy management strategies specifically designed for interdependent services. When a server drops its processor frequency to conserve battery in a complex application, it does not merely delay a single task. Instead, it cascades delays across the entire DAG, potentially violating strict SLAs for the whole processing pipeline. Consequently, the current state of the art lacks a robust and online strategy that jointly manages service scheduling and dynamic resource allocation for complex applications operating under the constant threat of total energy depletion. This specific void motivates the core contribution presented in Chapter 6.

3.4 Evaluation Environments and Tools

To evaluate the performance of proposed service placement solutions, researchers utilise various tools to perform extensive experiments. These evaluations are conducted in targeted environments, prefer-

ably with setups that closely mimic realistic conditions. In this subsection, we classify the most common tools used in the literature, ranging from high-level simulators to real-world testbeds.

Simulation tools are ubiquitous in MEC research because deploying large-scale, complex mechanisms in real-life scenarios is often cost-prohibitive or logistically impossible. To accurately simulate the MEC environment, key characteristics such as complex network topology, heterogeneity, and user mobility must be modelled. The selection of a simulator should be based on the specific functionalities required by the research objectives.

CloudSim-based Simulators: Based on the well-known cloud simulator CloudSim, several edge-specific simulators have been developed to extend its capabilities to the network edge.

As a prominent extension of CloudSim, iFogSim [48] supports the placement of customised DAG applications and offers native functions for monitoring latency, network usage, and power consumption. It allows for the integration of custom power models. However, the simulator has historically limited the network topology to a hierarchical, tree-like structure. Furthermore, it has limitations regarding the flexible routing of data packets; typically, computing devices within the same hierarchical layer cannot directly interact, which limits its use in mesh-like edge scenarios.

EdgeCloudSim [134] is another widely used simulator based on CloudSim. It addresses some of iFogSim's limitations by providing a more realistic network model that handles mobility and network congestion more accurately. It supports the deployment of custom applications and provides rich features for task execution monitoring. While researchers have extended it to support energy-related functionalities, it lacks native, out-of-the-box support for complex inter-service dependencies compared to iFogSim.

Lightweight Discrete-Event Simulators: For scenarios requiring rapid prototyping or the simulation of massive numbers of nodes without the overhead of the Java Virtual Machine, lightweight Python-based simulators are often preferred.

Based on the process-based discrete-event simulation framework SimPy [90], YAFS [65] enables the performance evaluation of placement, scheduling, and routing strategies. It natively provides evaluation metrics including network delay, path latency, and response time. A key advantage of YAFS is its flexibility; it allows network topologies to be defined freely as complex graphs using NetworkX rather than strict trees. It is highly extensible, allowing users to easily characterise custom energy models and cost models for dynamic scenarios.

Network-Generic Simulators: When the research focus lies heavily on network protocol behaviour, packet-level interference, or 5G specificities, generic network simulators are employed.

As a gold standard for network simulation, NS-3(Network Simulator 3) [117] supports building MEC networks that follow international telecommunication standards. It allows the simulation of thousands of connected devices with high fidelity at the packet level. However, because it is built specifically for sophisticated network layer simulation, it lacks native, high-level abstractions for heterogeneous computing node configuration (e.g., CPU containers) and application-level task scheduling, often requiring significant custom development to model service placement.

Another tool, OMNeT++ [143], is a modular discrete event simulator often used in conjunction with the Simu5G framework [95] (formerly SimuLTE). Unlike general-purpose simulators, Simu5G specifically models the data plane and control plane of 5G New Radio interfaces, allowing for precise evaluation of MEC applications communicating via realistic 5G stacks. While highly accurate for latency and bandwidth modelling, integrating complex application-level logic and dependencies can be computationally expensive and complex to implement compared to CloudSim-based tools.

Real-world Testbeds and Emulators: To validate theoretical findings in realistic environments, physical testbeds and emulators are employed.

Realistic large-scale experimentation environments such as Grid'5000 (located in France) and FIT IoT-LAB provide researchers with remote access to large clusters of heterogeneous hardware, ranging

from high-performance servers to low-power sensors. Similarly, Chameleon Cloud (in the United States) offers a reconfigurable experimental testbed for cloud and edge computing research. These platforms allow for the deployment of real software stacks (e.g., Kubernetes, OpenStack) on bare-metal hardware.

Bridging the gap between simulation and hardware, tools like Mininet or ContainerNet allow researchers to create virtual networks of hosts on a single kernel. They allow the execution of real application code (inside Docker containers) while virtually emulating network constraints (bandwidth, delay) between them.

3.5 Critical Assessment and Research Gaps

The comprehensive review of the state-of-the-art literature reveals that while significant strides have been made in MEC resource allocation, the convergence of complex application architectures and constrained energy environments remains largely unresolved. The evolution of edge workloads from monolithic tasks to dependent service graphs introduces intricate spatial and temporal execution constraints. Concurrently, the push towards green computing and remote edge deployment necessitates reliance on stochastic energy harvesting.

Synthesising the reviewed domains exposes four critical research gaps that limit the practical deployment of modern edge systems:

1. Algorithmic Overhead and Topology Trade-offs in Infrastructure-Centric Placement: As surveyed in Section 3.2, while server-side energy optimisation in edge infrastructure has emerged as an active area of research, existing infrastructure-centric schemes face significant methodological limitations. The majority of studies either focus exclusively on mobile device battery preservation (E_{UE}) or rely on computationally heavy learning-based paradigms (e.g., DRL) and exact solvers to manage server power dissipation (E_{server}). Although learning-based approaches handle complex state spaces,

they incur substantial offline training times, high computational overhead, and convergence instability, making them difficult to deploy in latency-critical edge scenarios. Furthermore, lightweight heuristics in server energy literature often simplify workloads to linear service chains. Consequently, there remains a trade-off: existing literature lacks lightweight, deterministic heuristic methodologies capable of natively co-optimising general DAG completion latency and edge server dynamic power without incurring heavy computational or training overhead.

2. Uncoordinated Offloading Cascades in Hybrid Energy-Harvesting Online Scheduling: While dependency awareness is a standard system requirement across modern edge applications (Section 3.2), state-of-the-art online scheduling and cloud offloading frameworks in hybrid-powered edge environments (Section 3.3.1) predominantly evaluate placement decisions for independent task units. Applying isolated offloading logic to graph-structured service workflows creates severe dependency propagation delays. Specifically, offloading a single sub-component of an interdependent workflow to a remote cloud without evaluating upstream and downstream dependencies triggers massive, unaccounted communication overhead across network links. There remains a distinct lack of lightweight, online scheduling mechanisms capable of dynamically coordinating local edge server provisioning, selective cloud offloading, and inter-service communication overhead under fluctuating grid power prices and stochastic energy harvesting.

3. Absence of Depletion-Safe Mechanisms for Complex Workloads: In purely renewable, off-grid systems (Section 3.3.2), techniques such as DVFS are heavily relied upon to mitigate total battery exhaustion. However, existing literature primarily applies these frequency downscaling techniques to independent tasks. There is a profound gap in understanding the cascade effect where slowing down a single server processing a dependent task propagates delays across the entire DAG. The literature lacks a robust, online management strategy that mitigates battery depletion risks without violating strict SLAs for the entire processing pipeline.

4. Lack of Flexible, High-Level Simulation Tooling: Finally, as highlighted in Section 3.4, current

evaluation environments struggle to support this specific intersection of challenges. CloudSim-based tools often enforce rigid hierarchical network topologies, while network-generic simulators like NS-3 lack the native application-level abstractions required for rapid, large-scale service placement evaluation. There is a need for highly extensible tooling capable of simulating both complex internal service dependencies and realistic, mesh-like MEC architectures.

3.6 Relation to Own Work

This dissertation directly addresses the critical gaps identified in the current literature. By abandoning the assumption of independent tasks and directly confronting the harsh realities of energy-constrained edge infrastructure, the subsequent chapters propose novel, dependency-aware resource allocation and scheduling strategies. The relationship between the identified gaps and the core contributions of this work is mapped as follows:

Addressing Gap 1 (Chapter 4): To address the computational overhead and structural limitations of existing server-side energy optimisation schemes, Chapter 4 introduces two novel deterministic heuristic algorithms (EDEM and DEM). Moving beyond heavy learning-based models and user-equipment-centric formulations, these algorithms provide fine-grained, low-overhead service placement for general DAG workloads designed to simultaneously minimise end-to-end latency and edge server dynamic energy consumption.

Addressing Gap 2 (Chapter 5): To mitigate communication cascades and uncoordinated offloading inefficiencies, Chapter 5 presents an online dynamic service scheduling mechanism (DSO) tailored for graph-structured workloads in hybrid energy-harvesting environments. By joint-optimising local server resource allocation, adaptive server scaling, and selective cloud offloading while enforcing precedence constraints, DSO minimises overall operational costs and promotes SLA compliance under fluctuating energy supply.

Addressing Gap 3 (Chapter 6): Confronting the extreme constraints of purely renewable systems, Chapter 6) presents a robust, depletion-safe energy management scheme (DSM). This contribution directly tackles the cascade effect missing in current DVFS literature, jointly managing service scheduling and energy allocation to enable complex service pipelines to meet strict deadlines without risking total server power failure.

Addressing Gap 4 (Methodological Contribution): To validate these strategies, this work extends the discrete-event simulator YAFS. As detailed in the methodology, this enhancement bridges the tooling gap by enabling the precise simulation of MEC architectures, complex DAG internal dependencies, and the integration of energy generation models.

Chapter 4

Energy-Latency Aware Dependent Service Placement in Cloud-MEC

4.1 Introduction

Driven by the rise of Fifth Generation networks and mobile edge environments, orchestrating complex applications across distributed edge infrastructures requires strategic resource allocation. Modern mobile applications rarely consist of isolated atomic tasks; rather, they comprise numerous interdependent microservices that collaborate to execute complex data processing workflows [163]. Represented as DAGs, these inter-service dependencies introduce intricate topological constraints where downstream services cannot commence execution until upstream precursors complete processing and forward their data payloads.

Existing service placement strategies in edge computing often struggle to effectively handle these complex, dependency-aware applications. While traditional methods recognise the importance of resource allocation, they frequently overlook the inter-dependencies of application modules or rely on rigid architectures, rendering them unsuitable for DAG-based scenarios. Furthermore, balancing mo-

mobile user satisfaction—specifically low end-to-end latency—against edge server operational efficiency and dynamic energy consumption presents a challenging MOO problem. To address this methodological gap, we propose fine-grained service placement approaches specifically designed for multi-tier Cloud-MEC environments. By strategically mapping dependent microservices across constrained edge MDCs and centralised cloud data centres, our approach minimises user makespan while actively mitigating server-side energy footprint. Since exact optimal placement in dynamic MEC environments is computationally intractable (NP-hard) [120], we employ a heuristic approach that maintains competitive performance against existing baselines while significantly reducing computational overhead.

This work addresses the service placement problem with precedence constraints among service applications. Our objective is to improve QoS, specifically by minimising latency, while optimising resource efficiency, measured by energy consumption and operational cost. The inherent complexity of these conflicting factors motivates our development of novel heuristic-based placement algorithms and an extended online resource scheduling strategy. Our contributions in this chapter are twofold:

- **Dependency-aware Service Placement Algorithms:** We propose two novel service placement algorithms specifically designed for the MEC environment. These algorithms consider precedence constraints between service components to optimise both end-to-end latency and dynamic energy consumption.
- **YAFS Platform Extension:** To support the evaluation of our algorithms, we developed an extension¹ to the YAFS simulation platform [65]. This extension enables the modelling of sequential task processing, a crucial aspect of service execution in MEC.

¹https://github.com/Sukiiichan/YAFS_MEC

Table 4.1: List of key notations for the system model and placement algorithms.

Indices	Description
$G = (M, L)$	Cloud-MEC network
M	Set of MDCs
S	Set of computing servers
L	Set of network connections
D	Set of data sources
γ_s	Memory capacity of server s
μ_s	Storage capacity of server s
f_s	CPU frequency of server s
b_l	Bandwidth of link l
$A = (V, E)$	Application graph
α_o	Memory requirement of module o
θ_o	Storage requirement of module o
$wl(o)$	CPU cycles required by module o to process one task
δ_e	Size of the transmitted data packet through edge e
T^{tran}	The transmission delay
T^{comm}	The communication delay
T^{exec}	The task execution time
$K(n)$	The multi-tenancy overhead coefficient of n modules
$EC(s)$	Dynamic energy consumption of server s
$EST(o)$	Earliest start time of module o
$EFT(o)$	Earliest finish time of module o
$EST(U)$	User-experienced latency

4.2 System Model and Formulations

In this section, we formally describe the service placement problem in the MEC system. The following models formalise the system variables, constraints, objectives, and decision variables. Table 4.1 presents the list of key symbols used in the formulation.

4.2.1 Key Assumptions

To formulate a tractable mathematical framework for dependency-aware service placement within the Cloud-MEC environment, our model adheres to several key operational assumptions:

- **Quasi-Static Infrastructure:** The physical network topology and edge server resource capacities remain quasi-static during the placement calculation window of an individual application DAG.
- **Deterministic Communication:** Network bandwidths and link propagation delays across inter-MDC backhaul and cloud connections are assumed known and stable during task execution. We abstract the communication channels as symmetric links.
- **Server-Side Energy Isolation:** Mobile user devices (UEs) only act as endpoints that initiate requests and receive processed output data; all intensive computing modules are deployed across edge servers or central cloud instances, isolating our energy minimisation focus directly to the infrastructure servers.
- **Steady-State System Stability:** We assume the MEC system operates in a stable regime. While transient queuing delays naturally occur at the service level and are captured during simulation, the mathematical formulation mitigates queue accumulation.
- **Non-Preemptive Execution:** Service module execution on an assigned server is non-preemptive; once started, a module runs to completion without interruption.

4.2.2 MEC Network Model

We consider a multi-tier Cloud-MEC network infrastructure (Figure 4.1) where MDCs act as resource pools at the edge, co-located with interconnected base stations. Within each MDC, two key entities exist: edge **computing servers**, available for service deployment and execution, and **data sources**, which generate data flows from databases, sensors, and IoT devices without performing local computations. Query requests from various **user devices** are handled concurrently by the MEC system. This multi-access edge network is further connected to a resource-rich cloud data centre.

We model the Cloud-MEC network topology as a graph $G = (M, L)$. Here, $M = \{m_1, m_2, \dots, m_n\}$

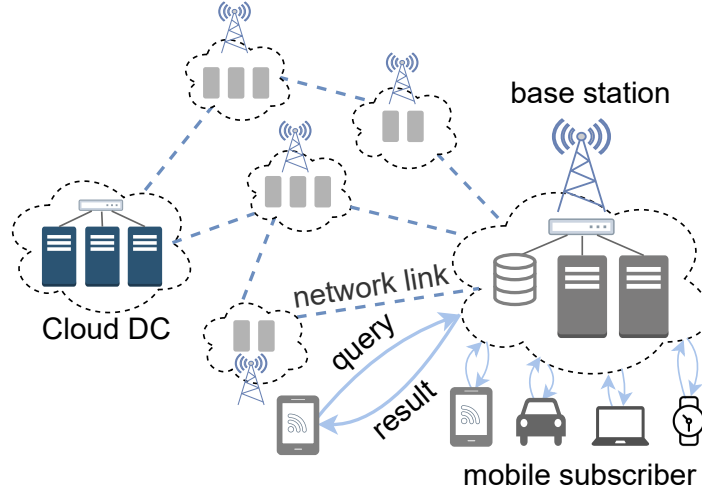


Figure 4.1: An example Cloud-MEC network consisting of multiple MDCs and one cloud data centre, serving multiple end users.

denotes the set of n MDCs and the cloud data centre, while $L = \{l_1, l_2, \dots, l_p\}$ represents the set of network connections between them. The set of computing servers maintained by MDC m_i is expressed as $S_i = \{s_{i,1}, s_{i,2}, \dots, s_{i,m}\}$, and $D_i = \{d_{i,1}, d_{i,2}, \dots, d_{i,k}\}$ indicates the data sources connected to the local network of m_i . The specifications for each server $s_i \in S$ are given by the tuple $\{\gamma_{s_i}, \mu_{s_i}, f_{s_i}\}$, representing memory capacity, storage capacity, and CPU frequency, respectively. For every network link $l \in L$, the connected MDCs are denoted as (l^{src}, l^{dst}) . The link is characterised by its bandwidth b_l and a constant propagation delay $prop_l$.

4.2.3 Application Model

Our processing model adapts the DAG model. Let $A = (V, E)$ represent the application graph, where the set of application modules is denoted by $V = \{D, O, U\}$, and the set of directed edges representing dataflow dependencies is $E = \{e_1, e_2, \dots, e_o\}$. $D = \{d_1, d_2, \dots, d_m\}$ represents the data source modules, while $O = \{o_1, o_2, \dots, o_n\}$ represents the service modules. The end receiver, referred to as U , is the UE.

Figure 4.2 depicts an example application DAG that comprises two data sources (a database and a

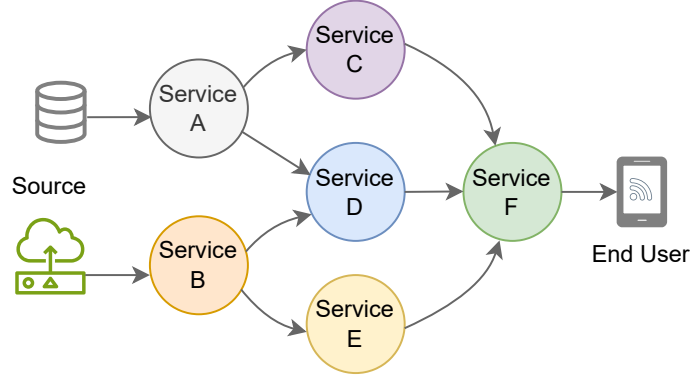


Figure 4.2: An example application graph consisting of two data sources, six dependent services, and one end user.

sensor), six service modules, and one end user. In this topology, Service A queries and processes data from the database, while Service B reads and processes data from the sensor. Both services then forward their results to downstream services for aggregation and analysis.

Each service module $o_i \in \mathcal{O}$ represents a processing operator or function with properties $\{\alpha_{o_i}, \theta_{o_i}, wl(o_i)\}$, denoting the memory, storage, and CPU cycles required to process one task, respectively. Modules with no predecessors that require input from external data sources are termed *entry modules*. Conversely, modules without successor nodes are termed *exit modules*; the outputs of all exit modules are directed to the end user. To clarify the operational distinction among services, we categorise service modules into **critical services** and **non-critical services**. Critical services reside directly on the application's critical path—the sequence of dependent modules incurring the longest cumulative delay from entry to exit—meaning any slowdown in their execution directly increases the end-to-end user latency. Non-critical services operate on parallel branches with time slack, allowing their execution to be scheduled more flexibly to conserve energy without violating the application completion deadline.

Each edge $e \in E = \{e_1, e_2, \dots, e_o\}$ possesses an attribute δ_e representing the size of the transmitted data packet. The start node e^{start} is the source module sending data, and the end node e^{end} is the destination.

Our model adheres to the following assumptions: (i) a service is triggered only after receiving inputs from all predecessors; (ii) each task represents an indivisible, minimal unit of work; and (iii) a service node transmits results to its successors only upon establishing task completion.

4.2.4 Problem Formulation

Communication and Computation Model

The transmission delay T^{tran} for a data packet pkt through network link l is calculated using the following formula:

$$T^{tran} = \frac{size(pkt)}{bw(l)} \quad (4.1)$$

The communication delay during this process is composed of the transmission delay and propagation delay:

$$T^{comm} = \frac{size(pkt)}{bw(l)} + prop(l) \quad (4.2)$$

For a directed edge $e = (o_i, o_j) \in E$, when the upstream start module o_i is offloaded to server s_i and the downstream destination module o_j is deployed on s_j , the communication delay of this directed edge is composed of the cumulative transmission and propagation delays across each network hop required to forward a data packet pkt :

$$T^{comm}(o_i, o_j) = \sum_{l \in path(s_i, s_j)} \left(\frac{size(pkt)}{bw(l)} + prop(l) \right) \quad (4.3)$$

where the routing path is denoted as the shortest ordered sequence of physical network links connecting server s_i to server s_j :

$$path(s_i, s_j) = \{l_1, l_2, \dots, l_n\} \quad (4.4)$$

Additionally, between two services deployed in the same MDC, we assume a constant network delay. When computing server s_i runs at frequency f_{s_i} to process tasks, given a workload demand $wl(o)$

required by service module o , the task execution time is calculated using the following equation:

$$T^{exec}(o, s_i) = \frac{wl(o)}{f_{s_i}} \quad (4.5)$$

We also define the overhead coefficient for the multi-tenancy scenario of n service modules deployed on one server: $K(n)$. When n service modules including o are co-located by one server, the task processing time of o becomes:

$$T^{exec}(o|n) = T^{exec}(o) \cdot K(n) \quad (4.6)$$

For service modules with multiple predecessor nodes, the service module will not start the task execution until all its predecessors have finished their task processing and delivered the results to it. Therefore, for a service module o , the earliest time it starts task processing is decided by its most time-consuming predecessor. For o with the set of its predecessor $pred(o)$, we obtain its *Earliest Start Time (EST)* using the following formulation:

$$EST(o) = \max_{o_i \in pred(o)} \{EST(o_i) + T^{exec}(o_i) + T^{comm}(o_i, o)\} \quad (4.7)$$

Similarly, the Earliest Finish Time (EFT), which denotes the earliest time service module o finishes its task processing, is obtained as follows:

$$EFT(o) = EST(o) + T^{exec}(o) \quad (4.8)$$

And therefore the end-to-end latency experienced by user U can be expressed as $EST(U)$.

Energy Consumption Model

Given the supply voltage of the CMOS circuits v and the CPU operating frequency f , the dynamic power consumption of an edge computing server is expressed by the following equation:

$$P = kv^2f \quad (4.9)$$

where k represents a device-related coefficient. Here we assume that v and f are linearly dependent, therefore:

$$P = \beta f^3 \quad (4.10)$$

where β represents a device related factor.

We derive the dynamic energy consumption of each server s by calculating its dynamic power consumption over time, as expressed by the following equation:

$$EC(s) = \int P_s dt = \int \beta_s f_s^3 dt \quad (4.11)$$

During the processing of a type o task, if the CPU frequency remains unchanged, combining with 4.5, the total dynamic energy consumption can be approximated as follows:

$$EC(s, o) = \beta_s f_s^2 wl(o) \quad (4.12)$$

Placement Problem Formulation

To formally parameterise the placement decisions, we introduce a binary decision variable $x_{o,s} \in \{0, 1\}$ for each service module $o \in O$ and physical server $s \in S$:

$$x_{o,s} = \begin{cases} 1, & \text{if service module } o \text{ is deployed on server } s, \\ 0, & \text{otherwise.} \end{cases} \quad (4.13)$$

Our objective is to minimise user-experienced latency and overall dynamic energy consumption by determining an optimal mapping matrix $\mathbf{X} = \{x_{o,s}\}$. A placement plan is valid if it adheres to the following constraints:

(i) Every service module in O must be assigned to exactly one server in S :

$$\sum_{s \in S} x_{o,s} = 1, \quad \forall o \in O \quad (4.14)$$

(ii) The aggregated storage and memory resource usage on any physical server s must not exceed its hardware capacities μ_s and γ_s :

$$\sum_{o \in O} x_{o,s} \cdot \theta_o \leq \mu_s, \quad \text{and} \quad \sum_{o \in O} x_{o,s} \cdot \alpha_o \leq \gamma_s, \quad \forall s \in S \quad (4.15)$$

(iii) The MOO problem is formulated as simultaneously minimising user makespan and total infrastructure dynamic energy consumption:

$$\mathbf{Opt} : \min_{\mathbf{X}} EST(U), \quad \min_{\mathbf{X}} \sum_{s \in S} EC(s) \quad (4.16)$$

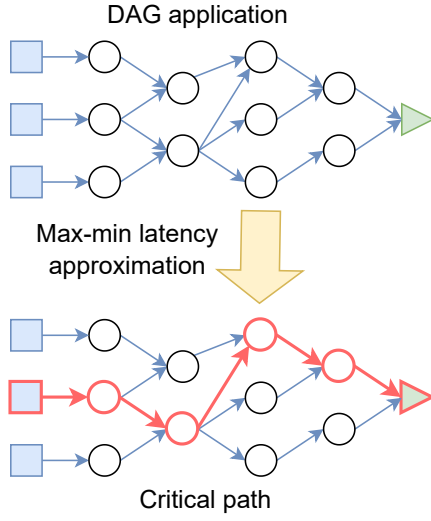


Figure 4.3: The coarse-grain stage of EDEM: bottom-up critical path selection. The algorithm iteratively approximates the *EST* for each service on selected servers, designating the predecessor that maximises the minimum *EST* as the critical node at each hierarchical level. The final critical path is highlighted in red.

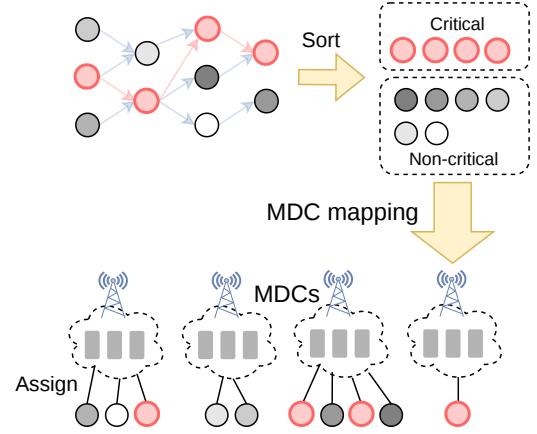


Figure 4.4: The coarse-grain stage of EDEM: service-to-MDC mapping. Red circles indicate critical modules along the identified critical path, whereas grey circles represent non-critical modules, with lighter shades denoting lower criticality. Black links indicate assigned placements between modules and target MDCs.

4.3 Algorithm Design

4.3.1 Energy-Aware Delay-Experienced Minimisation Algorithm

The proposed service placement algorithm, energy-aware delay-experienced minimisation (EDEM), adopts a two-stage approach to achieve a balance between latency minimisation and energy consumption reduction. The flow of the EDEM algorithm is illustrated in Figures 4.3, 4.4, 4.5. The coarse-grain stage (Figures 4.3 and 4.4) runs first, followed by the fine-grain stage (Figure 4.5).

In the coarse-grained stage, EDEM determines a service-to-MDC deployment plan that prioritises reducing end-to-end latency. Subsequently, the fine-grain stage refines the server deployment plan within each MDC, focusing on optimising energy consumption without affecting the overall latency.

Coarse-grain scheduler: Base on the critical path method, the coarse-grain scheduler seeks a bal-

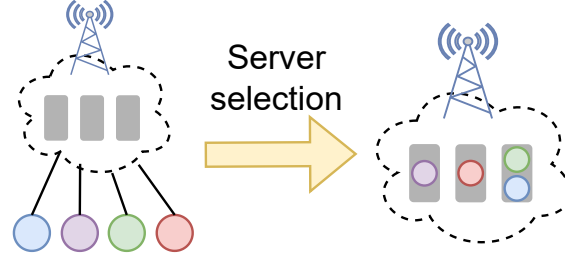


Figure 4.5: The fine-grain stage of EDEM: independent service-to-server mapping within each MDC.

Algorithm 1: Energy-Aware Delay-Experienced Minimisation (EDEM)**Data:** App. $A = (V, E)$, MEC $G = (M, L)$ **Result:** Server placement map $P : O \rightarrow S$ **1. Coarse-grain stage:**Initiate $S(A) \forall o \in O, \forall m \in M; CP = \emptyset;$ Explore $S(A)$ using post-order traversal:**for** $o.pred \in o.predecessors$ **do** **for** $m \in M$ **do** Compute $EST(o.pred, m);$ **end** Select $(o.pred, m)$ with $\min EST(o.pred, m);$ **end** $CP \leftarrow (o.pred, m)$ with $\max(EST(o.pred, m));$ Compute $Criticality(v);$ **2. Fine-grain stage:****for** $(o, m) \in critical_path$ **do** Initiate $S(O) \forall s \in m.servers;$ **for** $s \in m.servers$ **do** **if** $EST(o, s) \leq Criticality(o)$ **then** Compute $EC(o, s);$ **end** Assign v to s with $\min EC(o, s);$ Update load status of $s;$ **end****for** $(o, m) \notin critical_path$ **do** Initiate $S(O) \forall s \in m.servers;$ **for** $s \in m.servers$ **do** **if** $EST(o, s) + T_{comm}(o, o.succ) \leq Criticality(o.succ)$ **then** Compute $EC(o, s);$ **end** Assign o to s with $\min EC(o, s);$ Update load status of $s;$ **end**

anced configuration with minimal transmission latency and maximised processing efficiency, leading to the lowest approximated end-to-end latency. The pseudo-code of the coarse-grain scheduler is presented in Algorithm 1.

We define $S(A)$ as the solution space of placement options for all service modules in application A . The scheduler traverses $S(A)$ in topological order to determine the EST value for each module. Pursuant to Eq. (4.7), computing $EST(o)$ for any service module o requires the completion times of all its immediate predecessors, $o.predecessors$.

The critical path selection operates via a optimistic dynamic evaluation scheme, as illustrated in Figure 4.3. The algorithm proceeds level-by-level from the entry nodes (interfacing with data sources) to the final exit node. During the placement evaluation phase, we define $EST(o, m) \triangleq EST(o \mid place(o) = m)$ as the conditional start time of module o evaluated under the candidate assignment to MDC $m \in M$. For each visited module o and each candidate MDC $m \in M$, $EST(o, m)$ is evaluated assuming o is assigned to the least-loaded server $m.leastloaded$.

To determine the critical path segment leading to module o , the algorithm performs a two-step max-min identification over its predecessors $u \in o.predecessors$:

1. Best-case MDC Selection: For each predecessor $u \in o.predecessors$, identify the target placement m_u^* that minimises its completion time:

$$m_u^* = \arg \min_{m \in M} EST(u, m) \quad (4.17)$$

2. Critical Predecessor Designation: Among all selected predecessor-MDC pairs (u, m_u^*) , designated as candidate paths, the predecessor u^* yielding the maximum estimated time is identified as the critical node:

$$u^* = \arg \max_{u \in o.predecessors} EST(u, m_u^*) \quad (4.18)$$

The path tracing through u^* constitutes the critical path branch. Subsequently, for all non-critical

modules o' at the same DAG hierarchical level, criticality is quantified by the EST margin $\Delta EST = EST(u^*, m_{u^*}^*) - EST(o', m_{o'}^*)$. This procedure iterates until reaching the exit node.

The coarse-grained stage of the algorithm proceeds as illustrated in Figure 4.4. Critical modules are indicated by red circles, whereas non-critical modules are marked in grey. The previous phase establishes the critical path, aligning with the MDC allocation plan for the critical modules.

At this point, non-critical modules remain unassigned to MDCs. Subsequently, the load status of the MDCs participating in the critical path allocation plan is updated, along with the least-loaded server $m_{leastloaded}$ within each MDC $m \in M$. The max-min procedure then resumes, allocating non-critical modules until their MDC assignments are finalised. Once all service modules are mapped to their respective MDCs (indicated by black links in the figure), the algorithm proceeds to the module-to-server allocation phase.

As defined previously, at each bottom-up hierarchical level in the DAG, the critical node is the one with the maximum approximated EST value. To establish the placement priority among non-critical modules, we define the criticality of a non-critical module by the difference between the EST value of the critical module at its upper hierarchy level and the minimum EST value of its successor modules. A module with a smaller time difference is considered more critical (represented by darker shades) and is therefore prioritised in the subsequent stage.

Fine-grain scheduler: Following the determination of the service-to-MDC placement plan during the coarse-grained stage, for each MDC involved, a fine-grain scheduler runs to obtain the module-to-server placement solution. As can be seen in Figure 4.5, in this stage, the server selection for service modules is performed independently within each MDC. The fine-grained scheduler prioritises placing modules on less power-consuming servers while constraining the end-to-end latency of critical nodes to remain unaffected. Critical modules will take priority in placement, followed by non-critical modules. The pseudo-code of the fine-grain scheduler is presented as Step 2 in Algorithm 1.

Similar to the design of the coarse-grain scheduler, the fine-grained scheduler thoroughly evaluates all service-to-server placement options. For each module and every available server within the local MDC, the scheduler estimates the earliest start time. We use $EST(o, s)$ to denote the value of $EST(o)$ when service o is deployed on the server s . Subject to the constraint that the EST value of modules on non-critical branches must not exceed the EST of critical modules within the same hierarchical level, we evaluate the EST value $EST(o, s)$ for each potential placement option (o, s) . Placement options that violate this constraint are discarded, while the remaining options are deemed *valid*. From these valid options, the placement with the lowest energy consumption $EC(o, s)$, as approximated by Equation 4.2.4, is selected and incorporated into the placement plan. Upon finalising a service placement decision, the load status of the designated server is updated, and the estimated EST values of all affected placement options are recalculated. This iterative process continues until all service modules are assigned to specific servers.

4.3.2 Delay-Aware Energy Minimisation Algorithm

In contrast to EDEM, which prioritises latency minimisation, the proposed delay-aware energy minimisation (DEM) algorithm prioritises energy efficiency. DEM initially seeks an energy-saving server placement plan. Subsequently, a refinement stage fine-tunes this plan to optimise latency while strictly adhering to established energy constraints. DEM can be broken down into the 3 steps, in order as shown in Figure 4.6, 4.7, 4.8.

Step 1: Greedy initialisation.

This phase establishes an initial service-to-server placement plan. As shown in Figure 4.6, all available servers within the MEC network are first sorted in ascending order of their device-related energy consumption coefficients. This coefficient, defined in Eq. (4.12), quantifies the energy consumption of a server for a given workload.

Next, a level-order traversal of the application graph commences from the entry modules. In Fig-

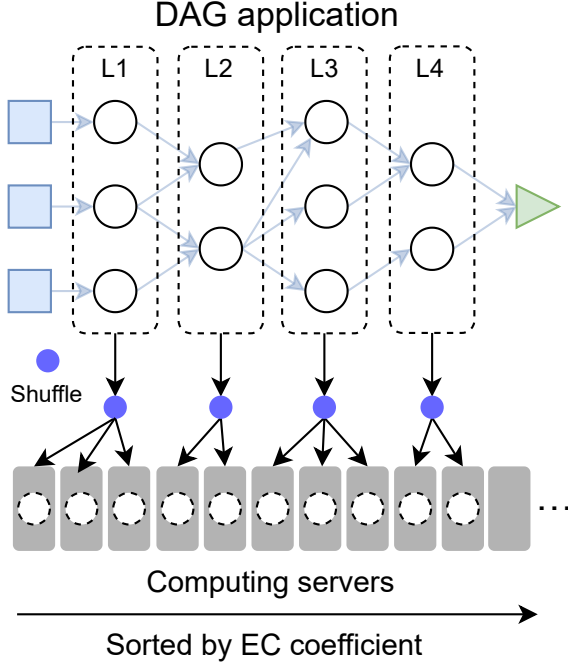


Figure 4.6: The phase one of DEM: a level-order traversal of the application graph to assign modules layer-by-layer to edge servers pre-sorted by their energy consumption coefficients.

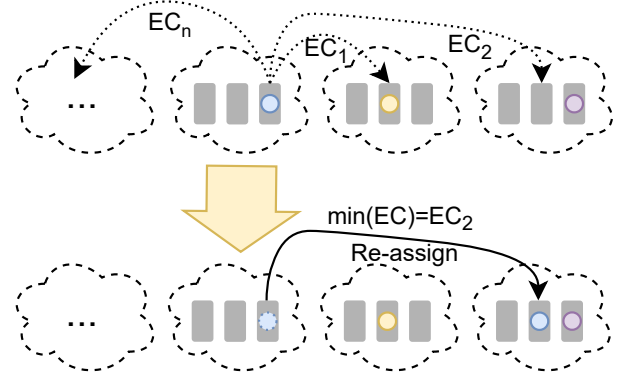


Figure 4.7: The phase two of DEM: iteratively reassign service modules to the least power-consuming servers across unexplored MDCs whenever energy consumption can be reduced.

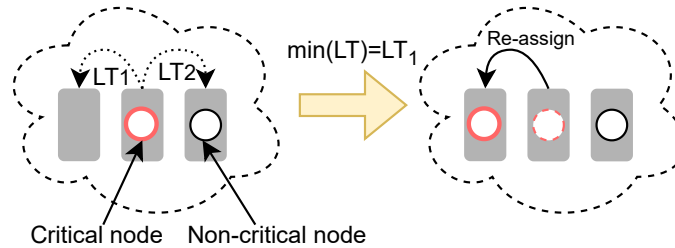


Figure 4.8: The phase three of DEM: iteratively reassign critical-path modules to alternative servers and recalculating end-to-end latency until no further reductions are possible.

ure 4.6, the services are split into four layers (indicated by dashed boxes L1–L4). At each layer of the traversal, the modules are randomly ordered (indicated by the blue 'shuffle' operation) and sequentially assigned to resources from the pre-ranked server list. Following this assignment, DEM estimates the EST value for each module using Eq. (4.7).

Step 2: Service-to-MDC placement alteration.

While *Step 1* prioritises energy-efficient servers, it does not automatically yield minimal total energy consumption. Geographically distant placements of interdependent modules can incur prolonged data transmission delays, forcing downstream servers to consume idle power while awaiting incoming packets. To mitigate this issue, a refinement stage iteratively evaluates candidate placements for each service module, as illustrated in Figure 4.7.

For a given module o , its current MDC assignment $m_{\text{curr}}(o) \in M$ from the initial phase is marked as explored, while all remaining MDCs form the set of candidate targets, denoted by $M_{\text{unexplored}}(o) = M \setminus \{m_{\text{curr}}(o)\}$. In Figure 4.7, potential reassignments are evaluated for the service highlighted in blue. Within each unexplored MDC $m \in M_{\text{unexplored}}(o)$, the server with the lowest energy consumption coefficient s_m^* is selected as the candidate host:

$$s_m^* = \arg \min_{s \in S_m} \beta_s \quad (4.19)$$

As indicated by the dotted arrows, the algorithm then evaluates the hypothetical total dynamic energy consumption $EC(o, m)$ resulting from reassigning module o to s_m^* :

$$EC(o, m) = \sum_{s \in S} EC(s \mid P(o) = s_m^*) \quad (4.20)$$

The candidate MDC m^* is chosen according to the following objective function:

$$m^* = \arg \min_{m \in M_{\text{unexplored}}(o)} EC(o, m) \quad (4.21)$$

The placement plan updates if and only if the candidate yields an energy consumption lower than the current configuration ($EC(o, m^*) < EC_{\text{current}}$), after which m^* is added to the set of explored MDCs for module o . Finally, the refined placement's overall latency $EST(U)$ and total energy consumption $\sum_{s \in S} EC(s)$ are recorded for the subsequent stage.

Algorithm 2: Delay-Aware Energy Minimisation (DEM)

Data: Application $A = (V, E)$, MEC network $G = (M, L)$

Result: Server placement map $P : O \rightarrow S$

1. Server sorting and initial service assignment:

Sort all servers $s \in G$ by $coeff(s)$;

$curServerIdx \leftarrow 0$;

Group service from level n to 0 and shuffle each set $S_n, \dots, S_0$

for service $o \in S_n, \dots, S_0$ **do**

 Assign o to server with index $curServerIdx$;

$curServerIdx++$;

end

for $o \in O$ **do**

 Compute $EST(o, P(o))$ using Eqs;

end

2. Energy-aware reassignment:

for $o \in O$ **do**

for $m \in M$ **do**

if $P(o) \notin m$ **then**

$s = \arg \min_{s^* \in m.\text{servers}} coeff(s^*)$;

if $EST(o, s) < EST(o, P(o))$ **then**

 Assign server s to o ;

end

end

Compute $EST(U)$ using Eqs;

Sum up $EstimatedEC(o, P(o)) \forall o \in O$;

3. Latency-aware reassignment:

Identify *critical_path* of A under placement P ;

for $(o, P(o)) \in \text{critical_path}$ **do**

for $s \in P(o).\text{mdc.servers}$ **do**

if $EST(U) > EST(U | P(o) = s)$ **then**

 Assign server s to o ;

 Re-identify *critical_path* of A ;

end

end

Step 3: Service-to-server placement fine-tuning.

Building upon the refined resource allocation from *Step 2*, this phase of DEM focuses on reducing

user-experienced latency, as illustrated in Figure 4.8. Based on the current placement plan, the algorithm identifies the application's critical path and critical services using the estimated *EST* values, isolating the modules that govern end-to-end latency.

In Figure 4.8, critical nodes are highlighted with red circles, whereas non-critical nodes are marked with dark circles. For each critical module, potential reassignments to candidate servers are evaluated to reduce critical-path latency. As indicated by the dotted arrows, the end-to-end latency is re-estimated for each candidate configuration and compared against the baseline assignment plan. If a candidate server s^* yields a lower overall latency, the service is reassigned to this server according to the following expression:

$$s^* = \arg \min_{s \in S} EST'(U \mid P(o) = s) \quad (4.22)$$

Following each reassignment, the critical path is dynamically recalculated to capture potential shifts in critical dependencies. This iterative process continues until no further latency reduction can be achieved. The resulting placement plan, balancing energy efficiency and latency, is then used to establish the final module-to-server mapping. The complete procedure is detailed in Algorithm 2.

4.4 Experimental Evaluation

4.4.1 Performance Indicators & Setup

We employed simulations to evaluate the performance of the proposed placement algorithms (DEM and EDEM). The YAFS fog simulator [65] was used and extended to support the sequential processing of dependent tasks. The edge network topologies were generated using the *NetworkX* library² based on three random graph models: Barabasi-Albert (B-A) [8], Watts-Strogatz (W-S) [149], and a ring topology.

²<https://networkx.org/>

To evaluate generalisability, we utilised real-world workloads from the Alibaba cluster trace dataset³, which provides DAG information for production batch workloads in large-scale clusters. To model user-submitted jobs that encompassing services, such as video stream processing, image recognition, machine learning inference, real-time analytics, and large-scale data preprocessing, we randomly selected 10 applications containing more than 10 modules. Module resource requirements were configured according to the traces.

Regarding temporal settings, tasks arrive periodically at a constant interval of 100 global time units, simulating continuous data flow from IoT sensors. The simulation duration for each experimental set was fixed at 20,000 time units, deliberately chosen so that the system transitions beyond the initial transient warmup phase and converges to a stable steady state, with captured performance metrics reflecting long-term system behaviour rather than initialisation anomalies. To generate statistically significant averages, each experiment was repeated five times under identical configurations. All simulations were conducted on a server equipped with four Intel Xeon Gold 6230N CPUs and 256GB of RAM, running the Ubuntu 20.04 operating system.

4.4.2 Performance Assessment

To evaluate the proposed service placement algorithms (DEM and EDEM), we constructed heterogeneous MEC-Cloud topologies comprising up to 20 MDCs and one cloud datacentre, utilising the three graph models previously mentioned. Each MDC housed up to five computing servers. Details regarding the specific MEC network configurations are provided in Table 4.2.

To model a heterogeneous computing environment, we configured distinct resource characteristics for cloud and edge servers. Cloud servers, representing powerful centralised resources, were assigned CPU frequencies of 3–5 GHz and 32–64 GB of RAM, reflecting common virtual machine instance types in public clouds (e.g., Amazon EC2, Azure, and Google Cloud). In contrast, edge servers fea-

³<https://github.com/alibaba/clusterdata>

	Cloud	MDC
CPU frequency (GHz)	[3,5]	[1,2]
RAM (GB)	[32,64]	[1,4]
Propagation Delay (ms)	8	1
Bandwidth (Gbps)	10	2

Table 4.2: MEC-cloud environment configurations.

tured more constrained resources (1–3 GHz CPU, 1–4 GB RAM), typical of edge hardware deployed closer to users under power and cost limitations, such as ARM-based platforms and compact x86 systems. Network connectivity also varied: the link to the cloud featured high bandwidth (10 Gbps) but higher latency (10 ms one-way delay), representing a Wide Area Network connection over large geographical distances. Conversely, edge network links offered lower bandwidth but minimal latency, characteristic of local access networks like 5G or LANs. These parameters were selected to create a realistic testbed contrasting centralised and edge capabilities.

The two algorithms were evaluated based on four key metrics:

- **Overall Energy Consumption (EC):** This metric represents the total power consumed by all servers during the execution period, estimated from CPU usage data according to Section 4.2.4.
- **Average User-Experienced Latency (LT):** This metric denotes the mean response time of user requests, calculated from raw timestamps recorded by the simulator during network transmissions.
- **Edge Prioritisation (EP):** This metric reflects the percentage of services deployed at the edge.
- **Algorithm Execution Time (ET):** This metric indicates the computational time required for the algorithm to generate a placement (i.e., the wall-clock time).

The performance of DEM and EDEM was compared against four state-of-the-art service placement algorithms: Response Time Aware (RTA) [13], Genetic Algorithm (GA) [122], Maximise Reliability Offloading (MROA) [126], and Energy-Makespan Multi-objective Optimisation (EM-MOO) [57].

We selected these algorithms for their varying levels of complexity, computational overhead, and optimisation goals. Comparing our proposed algorithms with these diverse alternatives provides a broader understanding of their relative strengths and weaknesses, enabling a more robust assessment of their overall performance and efficiency.

4.5 Evaluation Results

To evaluate the performance of the EDEM and DEM algorithm, a total of 8,100 experiments were conducted. For each of the 10 applications, 5 experiments were performed for each of the 6 algorithms. The number of MDCs (denoted by n) was varied across [5, 10, 20]. Similarly, the number of servers (denoted by m) within each MDC was varied across [2, 4, 8]. Furthermore, three different network topologies were utilised in the experiments. All figures presented in this section demonstrate the normalised performance of both *EC* and *LT* metrics. The base unit for each simulation time step is defined in milliseconds (ms). The source data emission interval is set to 100 ms to reflect the high-frequency continuous data streams generated by IoT sensors and mobile video feeds. The total simulated operational window is established at 20 seconds containing 200 complete task generation cycles. This duration provides sufficient time for steady-state queueing dynamics to stabilise across multiple consecutive DAG pipeline executions. Within this window, the performance curves of all algorithms are recorded and plotted across varying infrastructure scales.

Energy Consumption Comparison

In Fig. 4.9, we show the normalised energy consumption results for all algorithms across different network topologies. To understand the comparative trends in Figures 4.9, 4.10, and 4.11, it is essential to examine the underlying algorithmic behaviours of the baseline approaches:

- **RTA:** By operating as a purely greedy latency-minimisation heuristic, RTA consistently allocates tasks to the most computationally powerful high-frequency servers without considering

power scaling curves, resulting in extremely high overall energy consumption.

- **MROA:** Because MROA is explicitly designed to minimise UE energy, it aggressively offloads approximately 40% of service workloads directly to distant cloud data centres. While conserving UE power, this approach incurs massive backhaul network transmission overhead and heavy server-side dissipation, leading to the highest observed infrastructure energy footprint.
- **GA:** While GA searches the global combinatorial space effectively in small networks, as system scale expands (Figure 4.11), the exponential explosion of the solution space limits its population from converging to latency-optimal schedules within limited generations.
- **EM-MOO:** EM-MOO balances energy and delay moderately well in standard graphs, but falters under constrained topologies like the Watts-Strogatz network model due to rigid dependency mapping.

In contrast, DEM achieves significantly lower energy consumption compared to all other algorithms. DEM's Phase 1 initialisation explicitly prioritises deploying microservices onto servers with the lowest dynamic power dissipation coefficients (β_s). Following DEM, EDEM also achieves highly efficient energy consumption levels comparable to GA and EM-MOO while maintaining robust stability across network variations.

Latency Comparison

Figure 4.10 illustrates the normalised latency performance across all algorithms. Note that all metrics are min-max normalised per scenario, setting the upper bound (1.0) to the worst-observed latency. GA consistently exhibits the worst latency performance across all network topologies. In the B_A and RING topologies, GA's normalised latency is concentrated near 1.0, causing its boxplot distribution to collapse onto the upper plot boundary.

Among the remaining algorithms, RTA, EDEM, and MROA consistently achieve lower latency val-

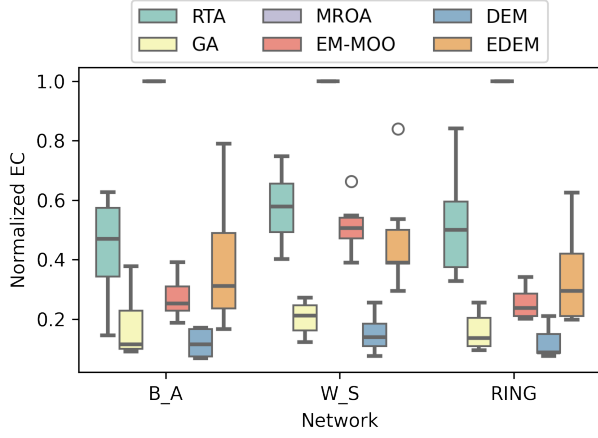


Figure 4.9: Normalised energy consumption (EC) across different network topologies with 20 MDCs and 4 servers per MDC.

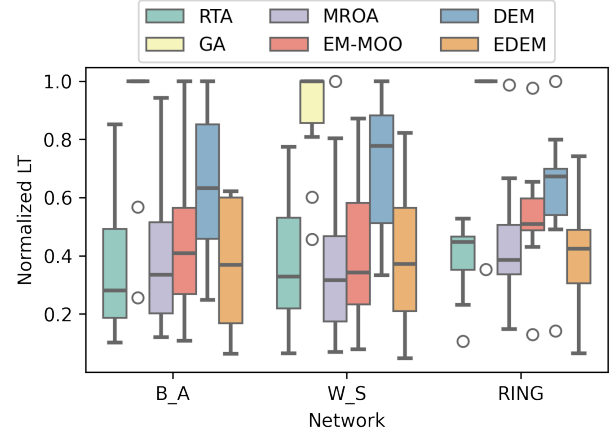


Figure 4.10: Normalised user-experienced latency (LT) across different network topologies with 20 MDCs and 4 servers per MDC.

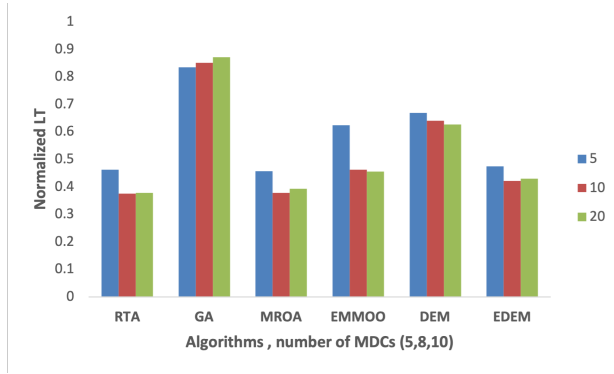


Figure 4.11: Normalised user-experienced latency (LT) with a varying number of MDCs ($n \in \{5, 10, 20\}$).

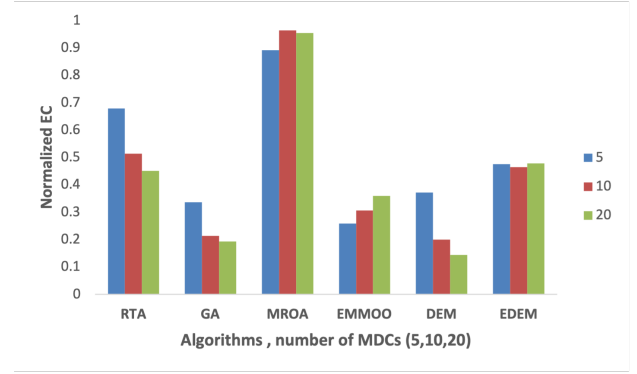


Figure 4.12: Normalised energy consumption (EC) with a varying number of MDCs ($n \in \{5, 10, 20\}$).

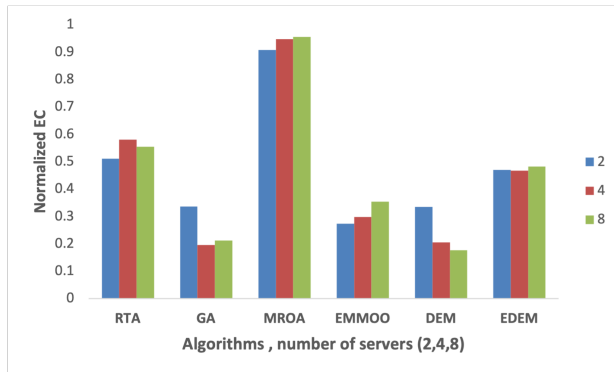


Figure 4.13: Normalised energy consumption (EC) with a varying number of servers per MDC ($m \in \{2, 4, 8\}$).

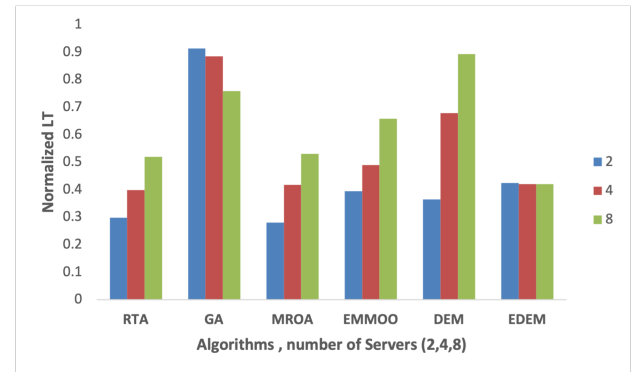


Figure 4.14: Normalised user-experienced latency (LT) with a varying number of servers per MDC ($m \in \{2, 4, 8\}$).

ues. However, RTA and MROA achieve these low latencies at the expense of significantly higher energy consumption, whereas EDEM successfully balances latency and energy efficiency.

Combining the results from Fig. 4.9 and Fig. 4.10, the distinct trade-offs of DEM and EDEM become clear: DEM strictly prioritises energy conservation, accepting higher latency as a consequence (ranking second-worst in latency performance, above only GA). In contrast, EDEM incorporates latency constraints, yielding superior delay performance—particularly in the RING topology where network latency presents a critical bottleneck.

Scalability Comparison

Figures 4.12 and 4.13 demonstrate the impact of scaling the number of MDCs ($n \in \{5, 10, 20\}$) and servers per MDC ($m \in \{2, 4, 8\}$) on overall infrastructure dynamic energy consumption. While both figures exhibit similar downward macro-level energy trends due to equivalent total capacity growth ($|S| = n \times m$), they evaluate fundamentally distinct operational dimensions: scaling n alters physical network topology and inter-MDC propagation delays, whereas scaling m increases local compute density per site without changing network path lengths.

Across both scaling dimensions, DEM and GA exhibit significant energy efficiency gains as capacity expands. For DEM, this reduction is driven by its global reassignment stage (Algorithm 2, Step 2), which actively evaluates candidate placements across unexplored MDCs to isolate host servers with lower power dissipation coefficients (β_s). Enlarging n or m expands DEM’s search space to identify low-power hardware. Similarly, for GA, a larger resource pool enables its fitness function to isolate lower-energy global schedules.

In contrast, EDEM demonstrates a highly stable energy consumption profile across both scaling scenarios. This stability stems from EDEM’s decoupled two-stage design (Algorithm 1): the coarse-grain stage first locks service-to-MDC assignments based on critical path latency constraints, after which the fine-grain stage optimises energy consumption strictly *locally* within each pre-selected MDC.

Because EDEM does not perform cross-MDC energy reassignments once MDC mappings are established, expanding n or m does not alter its baseline energy behaviour, making it a highly predictable option for infrastructure scaling.

When evaluating end-to-end latency scaling under varying MDCs (Fig. 4.11), both DEM and EDEM successfully reduce delay as n increases. Adding MDCs increases edge compute density closer to end users and data sources, mitigating backhaul transmission overhead. EDEM leverages this directly during its initial critical path allocation, while DEM achieves latency reductions during its Phase 3 refinement stage by re-allocating critical-path microservices to alternative servers to minimise $EST(U)$.

However, as the number of servers per MDC scales ($m \in \{2, 4, 8\}$, Fig. 4.14), DEM and EDEM exhibit a sharp behavioural contrast. While DEM achieves lower energy consumption under larger server pools, its user-experienced latency increases significantly. This performance trade-off stems from DEM's energy-first placement logic: as m increases, DEM isolates a wider selection of low-power servers (β_s) and disperses dependent microservice modules across a larger array of physical machines. While this dispersion maximises dynamic energy savings, it fragments the application DAG across host boundaries, compounding inter-service transmission and propagation delays (T^{comm}).

In contrast, EDEM maintains a virtually flat latency profile across all server configurations ($m \in \{2, 4, 8\}$). Because EDEM's coarse-grain stage binds critical-path modules to target MDCs under strict timing bounds prior to local server selection, it mitigates the cross-server fragmentation observed in DEM and GA. This stability establishes EDEM as the more reliable choice for latency-sensitive deployments in dense edge sites.

Edge Prioritisation Comparison

Table 4.3 presents the percentage of services deployed at edge servers, providing insights into the service placement strategies of different algorithms. The results of MROA align with our analysis, demonstrating a preference for non-local, powerful machines. Consequently, around 40% of the ser-

	RTA	GA	MROA	EMMOO	DEM	EDEM
B-A	85.86	97.75	58.52	98.80	98.24	85.70
RING	76.61	89.79	57.90	99.66	95.19	84.24
W-S	76.85	89.69	60.40	99.13	94.04	84.46

Table 4.3: Edge prioritisation results (*EP*).

	RTA	GA	MROA	EMMOO	DEM	EDEM
B-A	45.35	4110.22	3.13	1077.88	224.91	138.63
RING	42.79	4475.50	3.37	1066.64	315.66	173.59
W-S	45.96	4407.99	3.15	1064.44	290.66	161.51

Table 4.4: Execution time in ms (*ET*).

vices are offloaded to the cloud, leading to the highest observed energy consumption. RTA and EDEM have similar preferences in resource allocation, deploying 75-85% of services at edge facilities and the remaining tasks in the cloud. Such approach achieves a well-balanced outcome in terms of both latency and energy consumption by minimising transmission and energy costs while leveraging cloud resources for computationally intensive tasks. In contrast, DEM, EMMOO, and GA exhibit a strong preference for edge resource utilisation, deploying over 95% of services at the edge. Prioritising edge placement leads to the lowest energy consumption, but as a consequence of the energy-latency trade-off, the task execution time may be slightly higher compared to cloud execution, potentially impacting user-perceived latency.

Computational Overhead Comparison

For latency-sensitive applications, placement algorithms must execute rapidly. Table 4.4 quantifies this computational overhead. As anticipated, GA incurs the highest computational cost (4s). This latency stems inherently from its population-based search mechanics, which require repetitive fitness evaluations and evolutionary operations across multiple generations. EMMOO exhibits the second-highest overhead (1s), constrained by the complexity of its iterative process.

Conversely, MROA records the lowest execution times across all topologies. However, this speed is achieved by pruning the search space and bypassing constraint checks for latency and energy when

strict deadlines are absent. RTA, by narrowly focusing on a single objective, similarly benefits from a reduced overhead to achieve the second-fastest execution times.

Most notably, the proposed EDEM and DEM algorithms maintain execution times comparable to the lightweight RTA. This indicates that their heuristic mechanisms successfully navigate the complex, multi-objective search space without introducing prohibitive computational bottlenecks. By pairing near-optimal placement performance with minimal execution delay, EDEM and DEM prove highly suitable for real-time, time-sensitive application deployments in practical MEC systems.

Edge Impact Analysis

We also conducted an additional set of experiments to assess the impact of a MEC environment. In these experiments, we assumed a setup comprising 4 MDCs, 1 cloud data centre, and a BA topology. The results show that, on average, latency is reduced by 46.8% and energy consumption by 32.9% when scheduling decisions select resources from the edge-cloud continuum instead of relying solely on cloud resources.

Discussion and Limitations

While DEM and EDEM demonstrate strong scalability and performance, certain methodological limitations must be acknowledged. A primary limitation of EDEM is its reliance on critical path approximation to manage the execution of complex DAGs. While this heuristic significantly reduces computational overhead and simplifies dependency tracking, it may oversimplify execution dynamics in applications characterised by high degrees of parallelism or highly non-uniform workloads. By disproportionately weighting the initial critical path, the algorithm risks under-allocating resources to parallel, non-critical branches.

While we assumed stable network connections with sufficient bandwidth in the experiments, in congested edge networks, these secondary branches could experience unexpected queuing or transmission

delays, dynamically shifting the actual critical path during runtime and potentially biasing the initial placement decisions. Consequently, for highly parallelised microservices, this approximation may yield suboptimal makespans compared to exhaustive search methods.

4.6 Conclusion

This chapter presented EDEM and DEM, two novel algorithms for service module placement in MEC networks that explicitly address the dependencies between service modules. EDEM is designed to prioritise energy efficiency while mitigating adverse impacts on user-experienced latency. Conversely, DEM achieves substantial energy reductions by leveraging an operational paradigm that permits a flexible trade-off with increased latency.

While the experimental evaluation demonstrated the advantages of DEM and EDEM over established heuristics like GA and MROA, the comparative analysis can be further expanded. Future work shall benchmark the proposed algorithms against state-of-the-art exact solvers, such as ILP formulations, to establish definitive lower bounds and precisely quantify the optimality gap. Furthermore, comparing these heuristics against emerging learning-based placement strategies will be critical to evaluating their relative adaptability in highly stochastic environments.

Finally, the current framework focuses on static users and environments, which is a standard baseline for stationary edge computing. Incorporating dynamic variables, such as user mobility and fluctuating network conditions, would significantly increase the model's overall applicability. Extending this framework to handle dynamic topologies would render the system highly suitable for mobile contexts, such as connected vehicular networks and mobile augmented reality.

Chapter 5

Dynamic Resource Scheduling for Dependent Services in Energy-Harvesting MEC

5.1 Introduction

While static placement strategies optimise initial resource allocation, deploying complex microservices across dynamic edge environments introduces ongoing runtime challenges. For latency-sensitive applications constructed from interdependent components, such as mobile augmented reality pipelines requiring response times under 100 ms, maintaining strict SLAs during execution is critical. However, edge mobile MDCs operate under constrained communication, computation, and storage capacities that experience continuous real-time load fluctuations.

Simultaneously, the edge computing industry is increasingly driven by imperative sustainability goals. To reduce data centre carbon footprints and reliance on conventional fossil-fuel power grids, modern MDCs incorporate renewable energy harvesting (EH) technologies (e.g., solar and wind) backed by localised battery storage systems [125]. However, the inherent stochasticity of ambient environmental conditions makes renewable energy availability highly unpredictable. This introduces a challenging

online scheduling tension: accelerating server processing frequencies ensures low end-to-end latency but rapidly depletes finite battery reserves, whereas aggressive energy conservation via DVFS risks SLA deadline violations. Furthermore, severe green energy deficits necessitate offloading microservice tasks to remote cloud data centres, shifting execution burdens to grid-powered facilities at the expense of backhaul delay and operational monetary fees.

To address this dynamic interplay between SLA adherence, renewable energy stochasticity, and hybrid power management, we extend our static placement frameworks into an online resource scheduling strategy. By decomposing end-to-end application deadlines into localised soft constraints for individual DAG modules, our approach continuously balances server frequency scaling and cloud offloading to minimise operational costs while maximising green energy utilisation.

5.2 System Model and Formulations

In this section, we formally describe the service placement and resource scheduling problem in the Cloud-MEC system with Energy-Harvesting-enabled. Table 5.1 shows the notations of key indices used in this section.

5.2.1 Key Assumptions

Our dynamic resource scheduling framework operates under the following structural and operational assumptions:

- **Hybrid Energy Infrastructure:** Edge MDCs are powered primarily by local ambient energy harvesting devices paired with battery storage, supplemented by remote grid-powered cloud instances when edge energy or computational capacities are exhausted.
- **Time-Slotted Online Execution:** System runtime is partitioned into discrete intervals Δt . Task arrivals, renewable energy replenishment, and scheduler adaptations (DVFS scaling or offload-

ing) occur at slot boundaries.

- **Sub-deadline Decomposability:** To maintain end-to-end application deadlines without continuous global re-computation, overall DAG makespan constraints can be validly decomposed into proportional, independent execution sub-deadlines for individual microservices.

5.2.2 MEC Network Model

Consistent with the network model proposed in the preceding chapter, we consider a MEC infrastructure comprising multiple MDCs situated at the network edge. Each MDC hosts multiple edge computing servers, while a powerful, centralised cloud datacentre is also available for task offloading. UE, smart devices, and sensors connect to the MDC's local network, continuously transmitting service requests and receiving processed results.

As illustrated in Figure 5.1, each MDC is equipped with energy harvesting devices that capture renewable energy from the ambient environment to power the computing servers. The harvested energy is stored in a local battery associated with the energy harvesting device, which serves as the power source for the servers. In contrast, the Cloud datacentre is powered exclusively by the conventional electrical grid.

Adhering to the notation established previously, let $M = \{m_1, m_2, \dots, m_n\}$ denote the set of n MDCs. The energy harvesting device powering a specific MDC m is denoted by EH_m , and the maximum storage capacity of its associated battery is expressed as $Capa(EH_m)$.

5.2.3 Application Model

We model the application as a DAG comprising modules that can be individually deployed onto computing servers. Consistent with the model in the previous chapter, the application A is represented as a DAG $A = (V, E)$, where $V = \{D, O\}$ is the set of nodes constituting the application, and E is

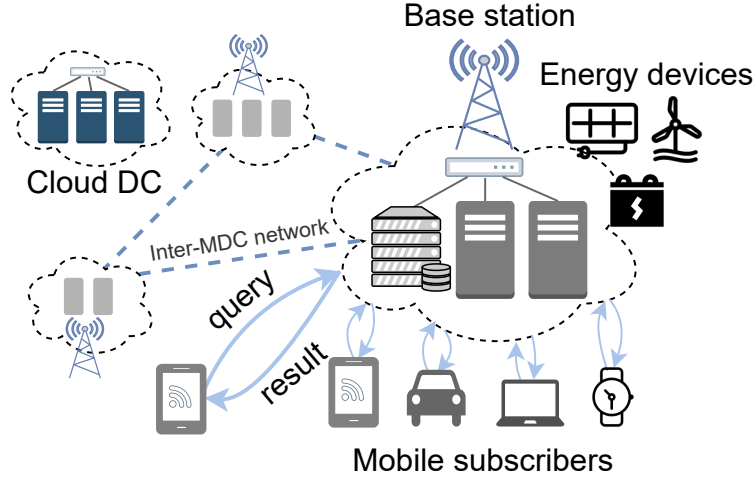


Figure 5.1: An example Cloud-MEC network with MDCs powered by energy harvesting devices.

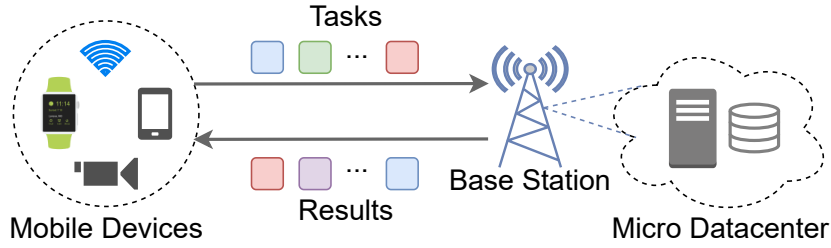


Figure 5.2: Tasks arrivals and responses in the system.

the set of directed edges representing the dataflow dependencies between these modules. Let $D = \{d_1, d_2, \dots, d_m\}$ represent the set of data sources involved in the application, and $O = \{o_1, o_2, \dots, o_n\}$ denote the set of service modules responsible for data processing operations. The end user is denoted by U .

As new source data arrives, the application DAG A is triggered to process tasks continuously. We define *rate* as the task arrival rate of application A . Furthermore, a deadline $deadline(A)$ is established to define the maximum acceptable completion time for the application tasks.

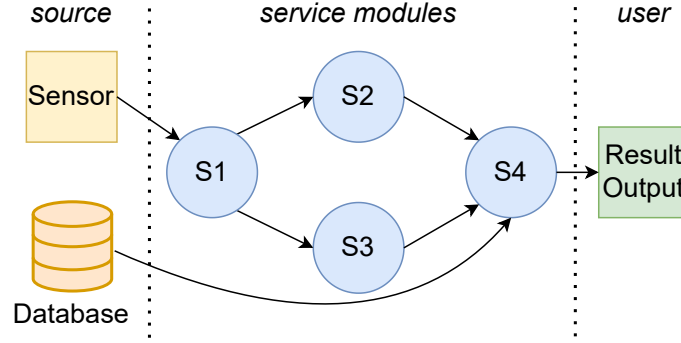


Figure 5.3: An example of dependent modules forming an application.

5.2.4 Problem Formulation

Communication and Computation Model

The calculation of task processing time, communication delay between service modules, and overall latency follows the definitions established in Section 4.2.4.

When deadline constraints of application A exist, each service request sent to the system is supposed to be finished in a time no longer than $deadline(A)$. In such situation, we define the term Latest Start Time (LST): the latest point in time a service module can start without delaying the overall application or violating the deadline, and similarly the Latest Finish Time (LFT): the time point that a service can complete its task processing such that the overall latency is not lagged. Therefore for any module o , we can obtain recursively:

$$LFT(o) = \begin{cases} deadline(A) & \text{if } o=U \\ \min_{o_i \in succ(o)} \{LST(o_i) - T^{comm}(o, o_i)\} & \text{otherwise} \end{cases} \quad (5.1)$$

and

$$LST(o) = LFT(o) - T^{exec}(o) \quad (5.2)$$

Energy Consumption Model

Consistent with the definitions provided in Section 4.2.4, we model the instantaneous power consumption of a server s as $P_s(t) = \beta_s(f_s(t))^3$. The total dynamic energy consumption for each server is then calculated by accumulating this instantaneous power consumption over time, using equation $EC(s) = \int \beta_s f_s^3 dt$.

Given this cubic relationship, dynamically scaling the CPU frequency f serves as the primary mechanism to minimise energy consumption, mathematically trading active power draw for increased execution latency.

This relationship can be demonstrated as follows. Consider a task with a fixed computational workload W (in CPU cycles). The time t required to complete this task is inversely proportional to the frequency ($t = W/f$). Consequently, the total energy EC consumed for this specific task is derived by integrating power over the duration of execution:

$$EC = P \cdot t = \beta f^3 \cdot \left(\frac{W}{f}\right) = \beta W f^2$$

As shown, the total energy consumption EC is quadratically proportional to the frequency ($EC \propto f^2$). Therefore, lowering the frequency results in a net reduction in energy consumption for the task, despite the linear increase in processing time. This convex relationship forms the basis of our DVFS strategy, allowing the scheduler to sacrifice latency for energy savings when deadlines permit.

Battery Model

We partition the time into a continuous series of time slices $1, 2, 3, \dots, n$, with equal lengths Δt . A new task may be generated in each time slice. We use $B_m(k)$ to represent the remaining power in the battery of the energy harvesting device EH_m at the beginning of the k th time slice.

During the k th time period, assuming that the operating frequency of each active server remains

constant, the battery state transition equation is derived as follows:

$$B_m(k+1) = B_m(k) + E_m^{harvested}(k) - E_m^{consumed}(k) \quad (5.3)$$

Also, the remaining power in the battery must satisfy the following at any time:

$$0 \leq B_m(k) \leq Capa(EH_m) \quad (5.4)$$

Use B_m^{ini} to denote the initial remaining power of battery B_m . Since MDCs purely rely on the harvested energy, combining with eq. 5.3, the energy stored in B_m at the beginning of time slot k is calculated as follows:

$$B_m(k) = B_m^{ini} + \sum_{n=1}^{k-1} (E_m^{harvested}(n) - E_m^{consumed}(n)) \quad (5.5)$$

Since the energy consumed by all MDC servers during the time slice cannot exceed the energy stored in the battery, the system satisfies the following inequality:

$$E_m^{consumed}(k) \leq B_m(k) \quad (5.6)$$

In scenarios where the harvested energy is insufficient to support the task processing during one time slice, the cloud offloading will be triggered.

Cloud Offloading and Cost Models

The operation of cloud data centres consumes energy drawn from the power grid. Let EC^{grid} represent the energy consumed from the grid, and let CI^{grid} represent the grid's carbon intensity factor. We calculate the resulting carbon footprint (CF) using the following formula:

$$CF = EC^{grid} \cdot CI^{grid} \quad (5.7)$$

If tasks belonging to service o are offloaded to the cloud, the financial cost to process a single task—given its specific workload demand $wl(o)$ and the cloud provider's unit pricing cp —is expressed as follows:

$$Cost_o^{cloud} = wl(o) \cdot cp \quad (5.8)$$

During the k -th time slice, let N_j^k denote the total number of task instances processed by service module $o_j \in O$. Cloud offloading for service module o_j is triggered dynamically when local edge battery reserves or computational thresholds are exceeded, incurring cloud provisioning fees. We introduce a binary decision variable $x_{i,j}^k \in \{0, 1\}$ to indicate whether the i -th task instance ($i \in \{1, \dots, N_j^k\}$) of module o_j during time slice k is executed on the cloud ($x_{i,j}^k = 1$) or processed locally at the edge ($x_{i,j}^k = 0$). By accumulating the monetary execution fees across all offloaded tasks, the total cloud expenditure during the k -th time slice is calculated using the following equation:

$$Cost^{cloud}(k) = \sum_{o_j \in O} \sum_{i=1}^{N_j^k} x_{i,j}^k \cdot wl(o_j) \cdot cp \quad (5.9)$$

Scheduling Problem Formulation

Our primary scheduling objective is to minimise total operational cloud fees while maintaining strict adherence to application deadlines and physical hardware limits. Formally, for time slice k , the opti-

misation problem P_2 is formulated as follows:

$$\begin{aligned}
\mathbf{P}_2 : \quad & \min_{\mathbf{X}^k, \mathbf{f}^k} \sum_{o_j \in O} \sum_{i=1}^{N_j^k} x_{i,j}^k \cdot wl(o_j) \cdot cp \\
\text{subject to:} \quad & T_{delay} \leq \text{deadline}(A), \\
& 0 \leq B_m(k) \leq \text{Capa}(EH_m), \quad \forall m \in M, \\
& f_s^{\min} \leq f_s(k) \leq f_s^{\max}, \quad \forall s \in S, \\
& E_m^{\text{consumed}}(k) \leq B_m(k), \quad \forall m \in M, \\
& x_{i,j}^k \in \{0, 1\}, \quad \forall o_j \in O, i \in \{1, \dots, N_j^k\}.
\end{aligned} \tag{5.10}$$

In Equation 5.10, $\mathbf{X}^k = \{x_{i,j}^k\}$ represents the binary cloud offloading decision matrix, and $\mathbf{f}^k = \{f_s(k)\}$ represents the continuous CPU operating frequency assignments across edge servers. The constraint $T_{delay} \leq \text{deadline}(A)$ constrains end-to-end user latency within agreed SLAs. The battery constraints require that instantaneous energy consumption $E_m^{\text{consumed}}(k)$ never surpasses current storage $B_m(k)$, nor does battery state exceed hardware physical capacity $\text{Capa}(EH_m)$. Finally, server frequency scaling is bounded within operational hardware thresholds $[f_s^{\min}, f_s^{\max}]$.

5.3 Algorithm Design

After integrating the energy harvesting technique with the MEC system, deadline constraints and energy budgets must be considered throughout the application execution.

Therefore, with the two offline placement algorithms in hand, we decided to design an online scheduling strategy that monitors the status of the energy-harvesting MEC system and continuously makes decisions for server operating frequency scaling and cloud offloading. Our online scheduling algorithm operates across three coordinated phases:

Table 5.1: Summary of key notations for the dynamic scheduling and offloading model.

Indices	Description
$G = (M, C, L)$	Cloud-MEC network
M	Set of MDCs
C	The Cloud datacentre
L	Set of network connections inside MEC
$bw(l)$	Bandwidth of network link l
$prop(l)$	Propagation delay of l
S_i	Computing servers in MDC m_i
f_{s_i}	Maximum CPU frequency of server s_i
cp	Unit price of cloud datacentre
D_i	Data sources in m_i
d_i	Single data source in D
EH_m	Energy harvesting device powering MDC m
$Capa(EH_m)$	Battery capacity of EH_m
$EC(s)$	Energy consumption of server s
$path(s_i, s_j)$	Routing path between two servers
B_m	Battery level of EH_m
A	Application graph
$S(A)$	State space of services in A
D	Set of data sources in A
O	Set of service modules in A
o	Single service module in O
$wl(o)$	Workload of single task processed by module o
E	Set of dependency edges in A
$pred(o)$	Set of predecessor modules of o
$succ(o)$	Set of successor modules of o
$deadline(A)$	Maximum task completion time of A
$sd(o)$	Sub-deadline of module o
$EST(o)$	Earliest processing start time of module o
$LST(o)$	Latest processing start time of module o
$LFT(o)$	Latest processing finish time of module o
$AST(o)$	Actual processing start time of module o
$AFT(o)$	Actual processing finish time of module o

Phase 1. Initial energy-latency-aware service placement. At the initialisation stage, we pass the edge network information and the application to the EDEM or DEM algorithm to generate a service-to-server assignment plan. Service modules will be deployed on MEC edge servers, after which application execution will begin.

Phase 2. Deadline constraints breakdown. Given the deadline constraint of the application, and with knowledge of the app graph structure as well as the task size and workload of every service, we calculate a sub-deadline for each module. Inspired by the deadline decomposition method from [84], we traverse all modules in a top-down order and determine the value of $LST(o)$ and $LFT(o)$ of each module $o \in O$ according to Eq. 5.1 and Eq. 5.2. Starting from the overall deadline and working backward, we use the upward rank $rank(o)$ to quantify the impact of the module to the entire schedule: the longer it takes to process sub-tasks and transmit data from the module to the end user, the higher the $rank(o)$ value becomes. Finally, The sub-deadlines of every module o , denoted as $sd(o)$, are calculated as proportions of $deadline(A)$.

The relations between the rank of a module o and its successor modules can be expressed as follows:

$$rank(o) = \max_{o_i \in succ(o)} \{rank(o_i) + T^{comm}(o, o_i)\} + T^{exec}(o) \quad (5.11)$$

Therefore the highest rank belongs to the entry source modules, which is calculated as follows:

$$rank_D = \max_{d_i \in D} \{rank(d_i)\} \quad (5.12)$$

and the sub-deadline of module o can be calculated as follows:

$$sd(o) = \frac{rank_D - rank(o) + \min_{s_i \in S} \{T^{exec}(o, s_i)\}}{rank_D} \cdot deadline(A) \quad (5.13)$$

These sub-deadlines indicate the maximum flexibility each service has before impacting the overall

schedule, providing room for further resource optimisation and scheduling.

Phase 3. Scaling and offloading decision making. The third phase is the online resource scheduling process, which runs continuously as the application actively receives new data. The scheduler monitors the system's real-time status, including the battery levels of energy-harvesting devices and the latency between service modules. We use $AFT(o)$ to denote the actual finish time of service module o , and use $\mathbb{E}(FT(o))$ to represent the expected finish time of o . According to the deadline constraints breakdown in Phase 2, the finish time satisfies the following inequality:

$$0 \leq \mathbb{E}(FT(o)) \leq sd(o) \quad (5.14)$$

When the actual finish time of a service o is earlier than its sub-deadline, slack time is available. This creates an opportunity to scale down the operating frequency of the edge server, reducing energy consumption within an acceptable range and avoid draining the battery. The service time needed for o to process a task can be calculated by the difference between its actual start time and actual finish time, expressed as follows:

$$T^{exec}(o, P(o)) = AFT(o) - AST(o) \quad (5.15)$$

We use $ratio(s_i)$ to denote the scaling ratio of the CPU operating frequency of server s_i , and combining Eq. 5.14 yields the following condition:

$$AFT(o) < \mathbb{E}(FT(o)) \leq sd(o) \quad (5.16)$$

Assume the finish time of service o equals the sub-deadline, regarding the theoretical minimum allowed operating frequency, the system satisfies the following relationship:

$$wl(o)/f^{min} = sd(o) - AST(o) \quad (5.17)$$

and so the theoretical minimum allowed value of the scaling ratio equals:

$$ratio(s_i)^{min} = \frac{f^{min}}{f(s_i)} = \frac{AFT(o) - AST(o)}{sd(o) - AST(o)}, P(o) = s_i \quad (5.18)$$

Taking the midpoint between the minimum and maximum scaling ratio we get:

$$ratio(s_i) = \frac{sd(o) + AFT(o) - 2 \cdot AST(o)}{2 \cdot (sd(o) - AST(o))}, P(o) = s_i \quad (5.19)$$

and the operating frequency of server s_i will be scaled down by $ratio(s_i)$.

Conversely, when the actual finish time of a service o exceeds its sub-deadline, or the battery of the local energy harvesting device is depleted, cloud offloading for the service module is triggered. The module will be undeployed from the MDC and redeployed to the cloud, at which point cloud billing will commence.

To provide a clear operational understanding of our runtime strategy, the step-by-step logic of the Dynamic Scaling and Offloading (DSO) framework is summarised in Algorithm 3. A detailed explanation of the pseudocode is provided below:

- **Static Placement Initialisation (Lines 1–4):** The scheduler invokes either DEM or EDEM to compute an initial static mapping $P : O \rightarrow S$, deploying microservice modules onto local edge servers based on graph topology and average network latency.
- **Sub-deadline Decomposition (Lines 5–11):** The scheduler performs a level-order traversal of the application DAG to establish Latest Start Times (LST) and Latest Finish Times (LFT). It calculates upward ranks $rank(v)$ to assign proportional soft sub-deadlines $sd(v)$ for every component module.
- **Battery Monitoring and Emergency Offloading (Lines 13–20):** At each continuous online scheduling slot, the algorithm evaluates the battery reserve $B(m)$ of every MDC. If an MDC's

Algorithm 3: Dynamic Scaling and Offloading (DSO)**Data:** Application $A = (V, E)$, $deadline(A)$, MEC network $G = (M, L)$ **Result:** Server placement map $P : O \rightarrow S$, scaling and offloading decisions**1. Initial service assignment using our algorithms:** $Algorithm(A, G) = P : O \rightarrow S$ **for** $o \in O$ **do** Deploy o to server $P(o)$;**end****2. Deadline constraints breakdown:**Explore A using level-order traversal:**for** $v \in A.current_level$ **do** Compute $LST(v)$ and $LFT(v)$;**end****for** $v \in V$ **do** Compute $rank(v)$ and $sd(v)$;**end****3. Server scaling and service offloading:****while** $A.active$ **do** **for** $m \in M$ **do** **if** $B(m) == 0$ **then** **for** $s \in m.servers$ **do** $O_{offload} \leftarrow \{o | P(o) = s\}$; **end** **end** **for** $o \in \{O - O_{offload}\}$ **do** **if** $AFT(o) - AFT(o.pred) > sd(o)$ **then** $O_{offload} \leftarrow \{O_{offload} + o\}$; **else** Compute $ratio(P(o))$; $P(o).scale_down()$ **end** **end** **for** $o \in O_{offload}$ **do** Undeploy o from server $P(o)$; Redeploy o to cloud C ; **end****end**

battery is fully depleted ($B(m) == 0$), all services currently deployed on its servers are immediately marked for emergency cloud offloading ($O_{offload}$) to mitigate service outage risk.

- **DVFS Scaling vs. SLA Offloading Evaluation (Lines 21–28):** For active edge services not yet marked for offloading, the scheduler compares actual module execution progress against allotted sub-deadlines. If cumulative delays exceed the soft sub-deadline ($AFT(o) - AFT(o.pred) > sd(o)$), the module is added to $O_{offload}$ to mitigate cascading SLA violations. Conversely, if execution finishes early with positive time slack, the server's CPU operating frequency is scaled down ($scale_down()$) via DVFS to conserve dynamic battery energy.
- **Cloud Migration Execution (Lines 29–32):** All service modules accumulated in $O_{offload}$ are dynamically undeployed from local edge servers and migrated to the cloud instance C , incurring cloud monetary execution fees while mitigating SLA violations.

Complexity Analysis

To address potential concerns regarding the computational overhead of continuous monitoring in real-time MEC environments, we provide a formal asymptotic complexity analysis of the DSO strategy. The algorithm is divided into an offline initialisation phase and an online monitoring phase.

In Phase 2, the algorithm decomposes the global deadline constraints by performing a level-order traversal of the application DAG. The computation of the LST , LFT , and individual sub-deadlines (sd) requires visiting each module and dependency edge. This yields a one-time, offline initialisation complexity of $\mathcal{O}(|V| + |E|)$, which does not impact real-time execution.

Phase 3 encompasses the online decision-making process. During each discrete monitoring interval, the scheduler evaluates the battery status of the energy harvesting devices across all MDCs and their respective servers. It then iterates over the active service modules to calculate execution slack times, determining whether to trigger DVFS or cloud offloading. Let N denote the total number of MDCs,

M_{max} denote the maximum number of computing servers per MDC, and V represent the total number of service modules. The computational complexity for each real-time monitoring cycle is strictly bounded by $\mathcal{O}(N \cdot M_{max} + V)$.

Because this time complexity scales linearly with the number of infrastructure nodes and service components, the heuristic deliberately avoids the exponential computational overhead typically associated with continuous, real-time optimisation solvers. Consequently, the monitoring and decision-making footprint imposes a mathematically negligible burden on the edge servers, confirming its practical feasibility for latency-sensitive energy-harvesting MEC deployments.

5.4 Experimental Evaluation

We employed simulations to evaluate the performance of the DSO algorithm. For this purpose, we utilised YAFS-MEC, our extended version of the YAFS simulator, which was further enhanced to incorporate models for energy harvesting devices and battery dynamics.

Similar to the previous chapter, we extracted job characteristics and execution traces from the open-source Alibaba Cluster Trace dataset, which records real-world production job deployments across large-scale clusters of over 4,000 servers. Specifically, we mapped the multi-instance task topologies from the trace directly to our application DAG structures, capturing natural workload variance in computational demands ($wl(o)$) and heterogeneous memory and storage footprints.

To evaluate the performance of our online resource scheduling strategy, unlike the experimental settings in the previous chapter, we constructed MEC-Cloud environments featuring energy-harvesting-enabled base stations. The network topology generation used the Barabasi-Albert (B-A) model, and each MDC is purely powered by an energy harvesting device with a battery storage. Detailed configurations of the energy-harvesting MEC environment are shown in Table 5.2, similar to Table 4.2, with specs commonly observed in cloud and edge computing scenarios. The algorithm receives the

	Cloud	Energy-Harvesting MDC
CPU frequency (GHz)	4	[1,2.5]
Propagation Delay (ms)	50	5
Bandwidth (Gbps)	10	1
Renting fee (\$ per sec)	0.01	N/A
Battery capacity (Wh)	N/A	10
Energy harvesting device charging rate (W)	N/A	1
Device power factor	10^{-27}	10^{-27}

Table 5.2: Energy-harvesting MEC-cloud environment configurations.

result from either the EDEM or DEM algorithm as the initial service placement plan and makes server frequency scaling or service offloading decisions dynamically.

We evaluated the algorithm based on five key metrics:

- Number of Processed Requests (TP): the amount of use requests that have been successfully processed, recorded after the receiving of each result at the user side.
- Average User-Experienced Latency (LT): the mean response time of user requests processed, extracted from the message timestamps stored in the simulation traces.
- The Acceptance Rate of Processed Requests (AC): the proportion of successfully processed requests that meet the deadline constraint, recorded after the arrival of results.
- Overall Energy Consumption (EC): the total power consumed by edge and cloud datacentres during the task processing, estimated from CPU usage history using 4.2.4.
- Cloud Renting Cost (CRC): the total fee charged by the cloud datacentre for task execution, calculated based on billing time.

To evaluate DSO-DEM and DSO-EDEM, we benchmarked their performance against three distinct control strategies: a) offloading, which forces all services to the cloud; b) scaling, which dynamically modulates CPU frequency to consume available execution slack prior to sub-deadlines; and c) none, a strict static baseline that enforces the initial service placement without subsequent adaptation.

For all scheduling variations, we assume that each datacentre is connected to a renewable energy source and is equipped with a battery storage. Green energy will be harvested to power the datacentre continuously. We aim to evaluate how different scheduling strategies impact user experience, energy usage and operational costs through our experiments.

5.5 Evaluation Results

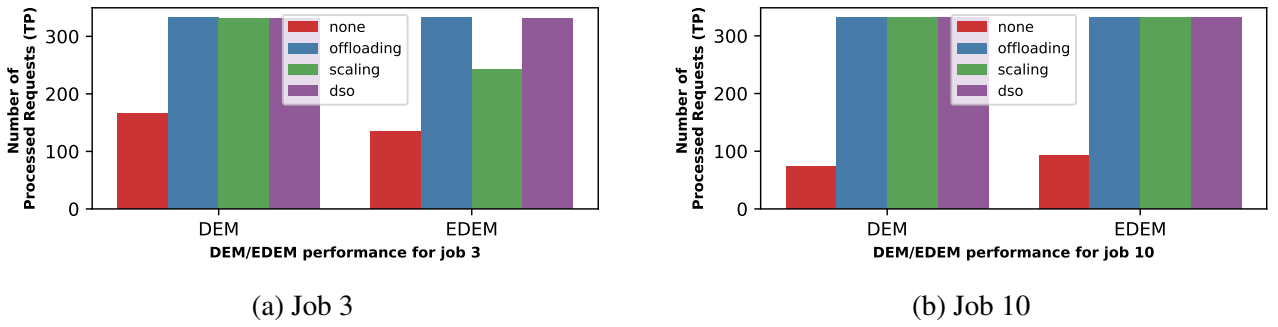


Figure 5.4: Comparison of total processed requests (TP) for Job 3 and Job 10 across different scheduling policies.

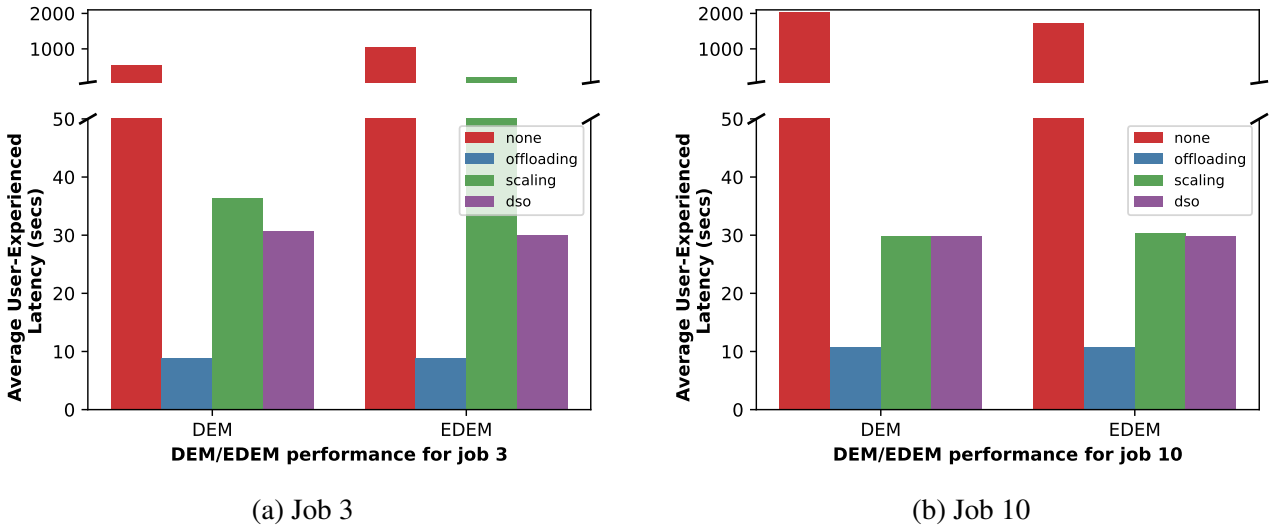
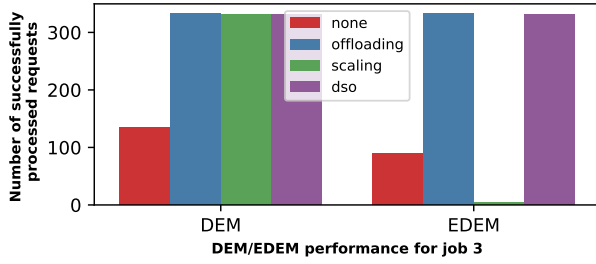
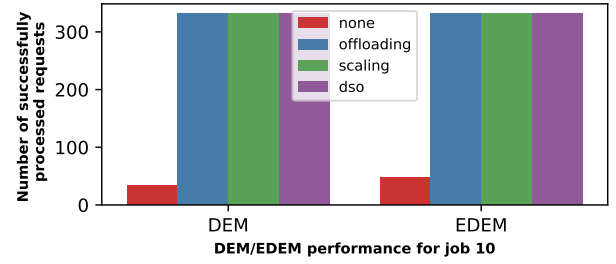


Figure 5.5: Comparison of average user-experienced latency (LT) for Job 3 and Job 10 across different scheduling policies.

Similar to the performance evaluation of the offline placement algorithms, we conducted experiments

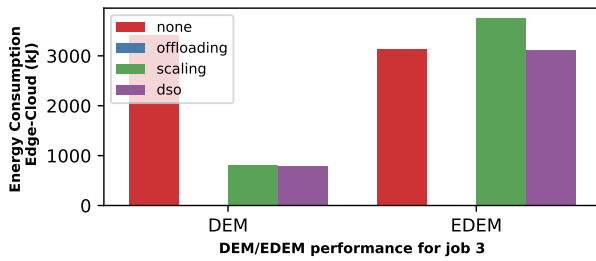


(a) Job 3

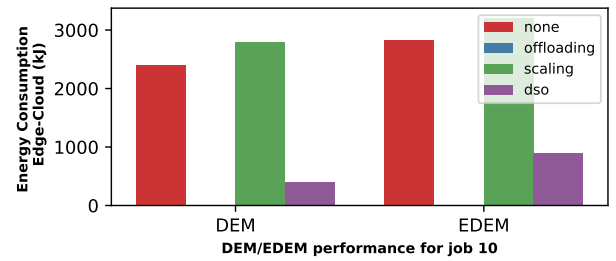


(b) Job 10

Figure 5.6: Comparison of the acceptance rate (AC) of processed requests for Job 3 and Job 10 across different scheduling policies.

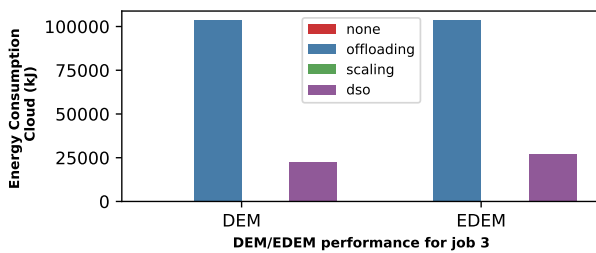


(a) Job 3

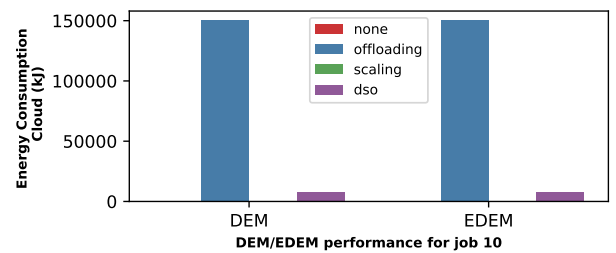


(b) Job 10

Figure 5.7: Comparison of overall energy consumption (EC) at the edge for Job 3 and Job 10 across different scheduling policies.



(a) Job 3



(b) Job 10

Figure 5.8: Comparison of overall energy consumption (EC) at the cloud for Job 3 and Job 10 across different scheduling policies.

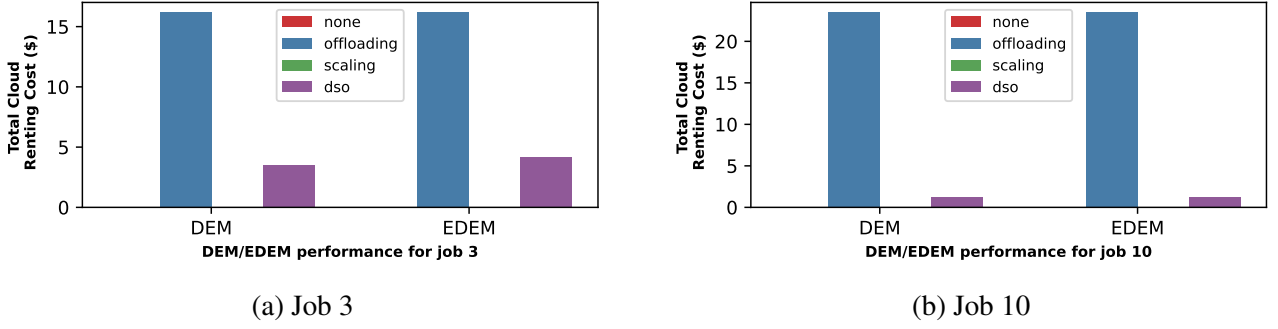


Figure 5.9: Comparison of cloud renting cost (CRC) for Job 3 and Job 10 across different scheduling policies.

for the four different scheduling algorithms using two distinct applications from the dataset. User requests were generated every 30 ms, and the schedulers were triggered at the same interval. The deadline for each application is pre-determined based on the method proposed in [84], with a deadline factor (γ) equals to 3.

System Throughput Comparison

Figure 5.4 demonstrate the total number of request processing completions. The results for application jobs 3 and 10 show that the baseline scheduler, labelled as *none*, can process less than half of the user requests received due to insufficient power supply at the edge. Under the same conditions, the scaling-only scheduler occasionally fails to process some requests but still achieves a significantly higher throughput than the baseline. The offloading-only scheduler, as expected, offloads all computation jobs to the cloud and successfully processes all user requests. Meanwhile, our DSO scheduler, labelled as *DSO*, achieves the same throughput level as the offloading-only scheduler.

Latency Comparison

Figure 5.5 demonstrates the average user-experienced latency of the processed requests. In both scenarios, the baseline scheduler results in extremely high latency due to the waiting time required for the batteries of the energy harvesting devices to recharge, causing task processing to be paused.

The offloading-only scheduler achieves the lowest mean latency by processing all tasks on powerful machines in the cloud datacentre. This is followed by the DSO scheduler, which strikes a balance between cloud offloading and edge execution. The scaling-only scheduler performs similarly to the baseline when it fails to address the battery shortage during the processing of job 3. However, in the remaining experiments, it achieves a mean latency slightly higher than the DSO scheduler.

Response Acceptance Comparison

Figure 5.6 illustrates the number of requests processed that meet the deadline. The results for the baseline scheduler reveal that without performing further actions, energy shortages not only reduce the throughput of request processing but also result in a low acceptance rate for responses. By scaling down the operating frequency of edge servers or offloading computations to the cloud, the system becomes more capable of processing requests within acceptable latency. However, the acceptance ratio for the scaling-only-EDEM case remains relatively low.

Initialisation Algorithm Tendencies

By analysing Figures 5.4, 5.5, 5.6 alongside the conclusions from the offline algorithm evaluations, the difference between scaling-only-DEM and scaling-only-EDEM when processing job 3 becomes clear. DEM prioritises energy saving, enabling it to perform better under energy constraints. In contrast, EDEM focuses more on processing requests quickly, which leads to poorer performance compared to DEM when energy supply is insufficient.

Energy Consumption Comparison

Figure 5.7 and Figure 5.8 show the total energy consumption resulting from the computations in the servers. Since the edge datacentres are entirely powered by energy harvesting equipment, only harvested energy is consumed by the baseline and scaling-only schedulers. The baseline scheduler, lacking any scaling policy, consumes a significant amount of edge energy while processing fewer

requests. As previously explained, the scaling policy dynamically adjusts the operating frequency of edge machines during slack time. This enables energy consumption to be significantly reduced while response times of requests remain within an acceptable range. Consequently, the edge energy consumed per request by the scaling-only scheduler is much lower than that of the baseline scheduler. The DSO scheduler, unlike the scaling-only policy, utilises cloud servers when necessary. As a result, it consumes additional energy by offloading part of the computations to the cloud. However, compared to the offloading-only scheduler, the energy consumption of the DSO scheduler is substantially lower.

Environment Impact Comparison

Next, we demonstrate energy savings in terms of carbon footprint reductions and for that we study the case of Job 3 using the results obtained from policies a) DSO and b) Offloading-only after applying DEM and EDEM algorithms (results were averaged).

During the 20000 timestamps simulated, our proposed strategy DSO resulted in an average energy consumption of 8.825 kWh, compared to 28.79 kWh consumed by the offloading-only approach under the same workload. This represents a grid energy saving of 19.97kWh. To quantify the environmental benefit, we use the average US grid carbon intensity factor of 0.37 kg CO₂e/kWh (based on EPA eGRID data for 2023[142]). This translates to an estimated carbon footprint reduction of 7.39 kg CO₂e achieved by our algorithm compared to the offloading-only strategy.

Cost Comparison

Figure 5.9 compares the cloud rental costs of the offloading-only scheduler and the DSO scheduler. The figure shows that the DSO scheduler incurs significantly lower operational costs while maintaining strong performance in both response latency and throughput. Although its average latency is slightly higher than that of the offloading-only scheduler, it still meets the deadline constraints and ranks as the second lowest in latency.

Discussion and Limitations

These evaluations demonstrate that the DSO scheduler effectively balances energy efficiency, performance, and cost. Unlike the baseline scheduler, which suffers from low throughput and high latency due to edge energy shortages, the DSO achieves full request processing throughput while successfully maintaining latency within deadlines. It outperforms the scaling-only approach by leveraging both cloud offloading and edge execution, and it consumes significantly less energy than the offloading-only scheduler. Furthermore, the DSO scheduler reduces cloud rental costs while delivering comparable latency and throughput. Ultimately, these results validate the DSO scheduler as a sustainable, cost-effective solution for managing workloads in energy-constrained environments.

While our results show that the proposed algorithms outperform state-of-the-art approaches and exhibit good scalability under increasing network complexity, it is important to acknowledge that these evaluations rely on relatively stable assumed conditions. Specifically, we assumed constant task arrival rates, typical energy harvesting availability, pre-known charging rates, linear updates, and average workload characteristics. Real-world deployments, however, frequently face extreme conditions such as sudden traffic spikes or prolonged energy shortages. Analysing these edge cases provides critical insight into the robustness of our solutions.

First, while the heuristic deadline decomposition defined in Eq. 5.11, Eq. 5.12, and Eq. 5.13 generalises well across structurally diverse DAGs by elegantly mapping critical paths to proportional sub-deadlines, its reliance on static metrics makes it vulnerable to bursty traffic. Because the policy fundamentally ignores queuing delays and resource contention, sudden traffic spikes can create bottlenecks. Although our algorithms can react quickly to sub-deadline violations by generating new placement decisions, backlogs can still accumulate during the time required for cloud offloading. This instantly consumes statically calculated slack and triggers cascading violations for downstream services. To address this, future work should transition from static assumptions to dynamic, queue-aware scheduling models. Incorporating real-time system load, adaptive backpressure mechanisms, and ser-

vice replication will be essential to maintain responsiveness and load-balancing in highly dynamic environments.

Second, prolonged periods of low renewable energy (e.g., due to adverse weather) coupled with low battery reserves pose a significant challenge. Under such conditions, the current algorithms prioritise cloud offloading to maintain service availability. While this preserves core functionality, it inevitably increases end-to-end latency and operational costs. This highlights the need for advanced energy prediction and more sophisticated energy management techniques. Integrating these capabilities as planned future work will enable our strategies to adapt more gracefully and efficiently to variable energy supplies.

5.6 Conclusion

This chapter integrated energy-harvesting techniques into the Cloud-MEC environment to design a dynamic, online resource scheduling strategy. This strategy effectively coordinates server frequency scaling and service offloading decisions while strictly adhering to stringent energy and latency constraints. Simulation results demonstrate that our scheduler successfully balances energy efficiency, operational cost, and overall performance, offering a sustainable and highly practical approach for managing user workloads in energy-constrained environments.

A promising direction for future research involves transitioning from reactive resource management to proactive planning by incorporating predictive models. The inherent instability of renewable energy sources, such as solar power, poses a continuous challenge for real-world deployments. Furthermore, real-world user demand is rarely constant; it is typically characterised by bursty, stochastic traffic patterns driven by user mobility and time-varying behaviours. Integrating user utility prediction would allow the system to anticipate these traffic fluctuations, enabling the scheduler to proactively reserve edge resources or adjust battery management strategies before congestion occurs. When combined

with energy forecasting based on historical weather and environmental data, this dual-predictive capability would significantly enhance the robustness and reliability of our solution in off-grid or highly variable scenarios.

Moreover, future work will explore integrating more advanced scheduling paradigms to handle the increasing complexity of MEC networks. While our current heuristic-based approach provides rapid, low-overhead placement decisions, comparing and potentially integrating it with reinforcement learning or stochastic optimisation frameworks could unlock further efficiency gains. For instance, reinforcement learning agents could continuously learn from and adapt to long-term environmental and traffic trends, while hybrid approaches could combine the fast execution of our current heuristics with the global optimality of stochastic methods. Finally, expanding the architectural model to incorporate edge-to-edge collaboration would allow tasks to be offloaded to neighbouring servers with surplus harvested energy. This approach could further mitigate reliance on the central cloud, minimise end-to-end latency, and enhance overall system sustainability.

Chapter 6

Online Energy Management and Task Scheduling in Harvest-Utility Mismatching MEC

6.1 Introduction

As established in preceding chapters, integrating ambient energy harvesting technologies (e.g., solar panels and wind turbines) is vital for sustainable, low-carbon edge computing [94, 156, 76]. While hybrid edge deployments can fall back on traditional electrical grids or remote cloud data centres during localised green energy deficits, there exist numerous critical edge scenarios where grid infrastructure or backhaul cloud connectivity is either physically unavailable, economically unfeasible, or intentionally disconnected.

Prominent examples include remote off-grid deployments in extreme environments, such as offshore oil platforms, agricultural monitoring meshes, and environmental sensor networks. Furthermore, during catastrophic natural disasters where central grids and wide-area backhauls collapse, standalone

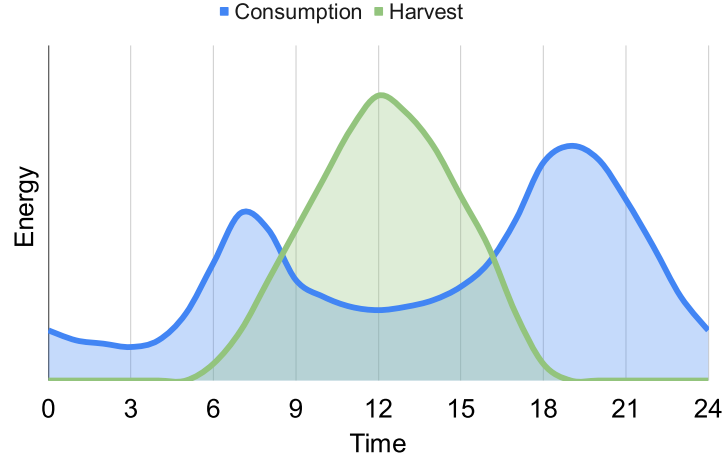


Figure 6.1: Temporal mismatch between energy harvesting and consumption over a 24-hour period.

edge servers must autonomously sustain critical communications. Finally, emerging zero-carbon initiatives mandate "green-native" 6G MDCs that operate entirely off-grid to achieve absolute carbon neutrality.

Operating in standalone off-grid environments introduces severe operational challenges driven by the stochastic nature of environmental energy replenishment [166]. Ambient solar and wind generation exhibits sharp temporal volatility and unpredictability. Concurrently, user request workloads fluctuate dynamically across daily peak and valley hours. This decoupling between non-stationary user demand and intermittent clean energy generation creates a distinct temporal mismatch between energy harvesting and consumption, placing immense strain on localised battery storage systems, as illustrated in Figure 6.1.

Unlike hybrid Cloud-MEC architectures where energy shortages are resolved by offloading tasks to grid-powered facilities, standalone off-grid MDCs lack a centralised fail-safe. Mitigating fatal battery depletion risks and catastrophic service outages requires a fundamental operational paradigm shift: transitioning from vertical cloud offloading to horizontal, edge-to-edge service migration.

To address this challenge, we propose a unified online resource scheduling strategy that jointly optimises DVFS alongside horizontal service migration for dependency-aware application workflows.

By periodically assessing system energy dynamics and filtering prediction noise via Double Exponential Smoothing (DES), our scheduler autonomously scales server operating frequencies and migrates microservice modules across peer MDCs to promote service continuity and low latency under strict off-grid energy constraints.

As highlighted in Gap 3 (Section 3.5), while DVFS is a widely adopted power management mechanism, naively applying frequency downscaling to independent tasks can cause severe execution delay cascades in graph-structured microservices. To overcome this limitation, our strategy in this chapter does not rely on static or uncoordinated downscaling. Instead, we employ a dependency-aware, bidirectional DVFS scheme: scaling CPU frequencies up during energy-surplus scenarios to reduce end-to-end latency, and downscaling conservatively during energy deficits to promote battery depletion safety without uncoordinated delay propagation.

6.2 System Model and Formulations

We consider a discrete-time system model where time is divided into decision epochs $t \in \{1, 2, \dots, N\}$ of constant duration Δt . At each decision epoch t , the system captures the current state and determine the configuration for the upcoming interval t . This approach follows the Time-Slotted Control paradigm. The key notations used in this chapter are summarised in Table 6.1.

6.2.1 Key Assumptions

Our standalone off-grid resource scheduling framework operates under the following structural and operational assumptions:

- **Fully Off-Grid Autonomy:** Edge MDCs are powered exclusively by local ambient energy harvesting devices paired with finite battery storage. No external electrical grid or remote cloud backhaul connectivity is available as a power or computational fail-safe.

- **Horizontal Service Migration:** Resource bottlenecks and impending battery exhaustion are mitigated strictly through horizontal microservice state migration across peer edge servers or neighbouring MDCs within the localised edge mesh.
- **Time-Slotted Prediction Filtering:** System runtime is partitioned into discrete control intervals Δt . At each interval boundary, DES filters forecast incoming workloads and solar/wind replenishment rates to guide proactive migration and DVFS adjustments.

6.2.2 MEC Network Model

Consistent with the system model presented in Chapters 3 and 4, we formulate the MEC network as a graph $G = (M, L)$. However, in this scenario, the system excludes both the cloud datacentre and the traditional grid energy supply. Figure 6.2 illustrates an example of our energy-harvesting MEC network, where each MDC is equipped with an energy harvesting device and a battery acting as its sole energy source.

We assume that any server s within an MDC m supports L_s discrete CPU frequency levels via DVFS. The set of available frequency levels is expressed as $\mathcal{F}_s = \{f_1, f_2, \dots, f_{L_s}\}$.

6.2.3 Application Model

The application model adopted in this chapter maintains the structure of a service dependency graph $A = (V, E)$, in which data flows from predecessor nodes to successor nodes. Once deployed, the application is activated by incoming user requests to process tasks.

In contrast to previous chapters, we assume the arrival rate of user requests is non-stationary. As user usage patterns fluctuate throughout the day, the application may experience periods of high-frequency triggering or remain idle for extended intervals. During the t -th time slot, the volume of request arrivals is denoted by $r(t)$.

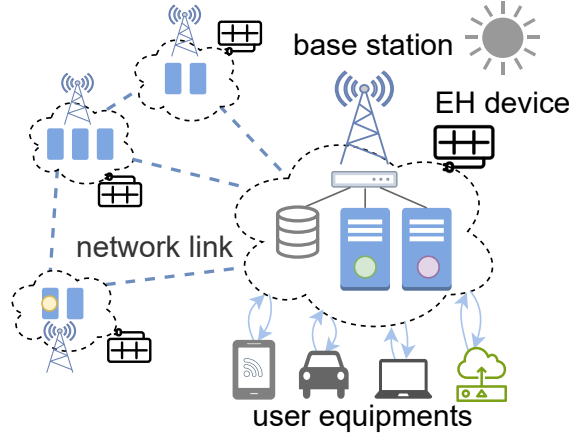


Figure 6.2: An example off-grid energy-harvesting MEC network with MDCs powered exclusively by renewable energy.

6.2.4 Energy and Battery Model

The battery models and the calculation of dynamic server energy consumption remain consistent with the definitions provided in the previous chapter.

For a specific MDC m during the t -th time slot, the aggregate energy demand is denoted by $E_{demand}(t)$. The total energy available for this period, denoted by $E_{supply}(t)$, is defined as the sum of the residual battery energy at the start of the slot and the new energy harvested during the interval. This relationship is expressed by the following equation:

$$E_{supply}(t) = B_m(t) + E_m^{harvested}(t) \quad (6.1)$$

In the event that an energy harvesting device's battery is depleted, all deployed services are suspended, and the MDC remains inactive until the battery recharges to a minimum safety threshold, denoted by B_{safe} . Consequently, we approximate the required recharge time using the following formula:

$$T_{recharge} = \frac{B_{safe} - B^{init}(t)}{\bar{P}_{harvest}(t, t + \Delta t)} \quad (6.2)$$

where $B^{init}(t)$ represents the current battery level, and $\bar{P}_{harvest}(t, t + \Delta t)$ denotes the projected average energy harvesting rate for the immediate future.

6.2.5 Migration Model

When migrating service modules from a source node to a destination node across the edge network, service interruption is inevitable. This latency arises as the active service pauses execution, encapsulates its execution state, transmits data across the backhaul link, and restores operation at the destination server. Let α_{o_i} denote the transfer volume of microservice o_i , encapsulating its memory state and container image footprint.

We define $T_{checkpoint}$ as the time required for state capturing, calculated as the product of a checkpointing coefficient (c_{chk}) and the service transfer volume α_{o_i} :

$$T_{checkpoint} = c_{chk} \cdot \alpha_{o_i} \quad (6.3)$$

The transmission delay, $T_{transfer}$, represents the network delay required to transmit the state data over the routing path links $l \in path$ with allocated bandwidth $bw(l)$:

$$T_{transfer} = \sum_{l \in path} \frac{\alpha_{o_i}}{bw(l)} \quad (6.4)$$

The restoration time at the destination node is modelled as follows, where c_{res} represents the restoration coefficient and T_{boot} is a constant startup overhead:

$$T_{restore} = c_{res} \cdot \alpha_{o_i} + T_{boot} \quad (6.5)$$

In addition to these phases, we account for a constant latency to cover network reconfiguration tasks, such as DNS propagation and routing table updates, denoted as $T_{overhead}$.

Consequently, the total migration latency T_{mig} is formulated as the sum of these sequential components:

$$T_{mig} = T_{checkpoint} + T_{transfer} + T_{restore} + T_{overhead} \quad (6.6)$$

6.2.6 Latency Model

Previously, we formulated the latency between two service modules as the summation of task execution time T_{exec} and communication time T_{comm} . In this chapter, we extend this formulation to account for service migration between edge servers and potential energy harvesting delays, where local services may need to pause execution to await battery recharging.

Consequently, we redefine the Earliest Start Time of any service o as follows:

$$EST(o) = \max_{o_i \in pred(o)} \{EST(o_i) + T^{exec}(o_i) + T^{comm}(o_i, o) + \mathbb{I}_{mig} \cdot T_{mig} + \mathbb{I}_{lb} \cdot T_{recharge}\} \quad (6.7)$$

where $\mathbb{I}_{mig}$ and $\mathbb{I}_{lb}$ are binary indicator functions that denote the occurrence of service migration and service suspension due to battery recharging, respectively.

6.2.7 Problem Formulation

In this section, we present an MINLP formulation for system cost minimisation, which jointly considers service migration and server DVFS. The objective function aggregates three key components: end-to-end latency, service migration overheads, and the penalty for battery outages.

Service Deployment Constraints

We define binary variables x_{ij} , $\forall o_i \in V$, $s_j \in S$ to denote whether service o_i is placed on server s_j or not, i.e.,

$$x_{ij} = \begin{cases} 1, & \text{if } o_i \text{ is placed on } s_j \\ 0, & \text{otherwise.} \end{cases} \quad (6.8)$$

Note that at anytime, to construct a valid placement of all services, a service o_i must be associated with one and only one server, i.e.,

$$\sum_{s_j \in S} x_{ij} = 1, \forall o_i \in V \quad (6.9)$$

Since the resources (i.e., memory and disk storage) of a server are limited, the total resource requirements of all services deployed at server s_j shall not exceed its capacity, yielding the following capacity constraint:

$$\sum_{o_i \in V} x_{ij} \alpha_{o_i} \leq \gamma_{s_j}, \forall s_j \in S \quad (6.10)$$

and

$$\sum_{o_i \in V} x_{ij} \theta_{o_i} \leq \mu_{s_j}, \forall s_j \in S \quad (6.11)$$

For each service module o_i during the t -th time slot, we define a binary migration indicator $m_i^{(t)} \in \{0, 1\}$ to denote whether the service has been migrated. This indicator is formulated as follows:

$$m_i^{(t)} = \frac{1}{2} \sum_{s_j \in S} |x_{ij}^{(t)} - x_{ij}^{(t-1)}| \quad (6.12)$$

where $x_{ij}^{(t)}$ represents the binary placement decision of service o_i on server s_j at time t .

Server Scaling Constraints

Regarding server frequency scaling, for each server s_k having L_s discrete frequency options, we define a binary variable y_{kl} to denote its frequency state:

$$y_{kl} = \begin{cases} 1, & \text{if } s_k \text{ operates at the } l\text{th frequency level} \\ 0, & \text{otherwise.} \end{cases} \quad (6.13)$$

Since a server can operate at only one frequency level at any given time, we impose the following constraint:

$$\sum_{l=1}^L y_{kl} = 1, \quad \forall s_k \in S \quad (6.14)$$

Energy Consumption Constraints

For any MDC m during a given time slot, the aggregate energy consumed by its servers must not exceed the available energy supply. This constraint is formulated as follows:

$$\sum_{s_k \in S_m} \left(\sum_{l=1}^L y_{kl} \cdot \beta_k f_l^3 \right) \Delta t \leq E_{supply}[m], \forall m \in M \quad (6.15)$$

Given the predicted task arrival volume for the time slot, denoted by $r(t)$, the total energy demand $E_{demand}(t)$ can be approximated as the product of the arrival volume and the energy consumption per task. This aggregate energy demand is expressed as follows:

$$E_{demand}(t) = r(t) \cdot \sum_{s_k \in S_m} \sum_{o_i \in V} \sum_{l=1}^L x_{ik} \cdot y_{kl} \cdot \beta_k f_l^2 \cdot wl(o_i) \quad (6.16)$$

System Cost Model

We define the total system cost as a weighted sum of the user-experienced latency, the migration overhead, and the risk penalty. The objective function is expressed as follows:

$$\min J = \omega_L \cdot LT(t) + \omega_M \cdot C_{mig}(t) + \omega_P \cdot P_{outage}(t) \quad (6.17)$$

where the average end-to-end latency $LT(t)$ represents the makespan of the application DAG processed during time slot t , determined by the EFT of the exit module across the critical path, defined explicitly as $LT(t) = \max_{o \in V} \{EST(o) + T^{exec}(o)\}$.

The migration cost, $C_{mig}(t)$, quantifies the system overhead and stability impact, formulated as follows:

$$C_{mig}(t) = \sum_{o_i \in O} m_i^{(t)} \cdot (c_{fixed} + c_{data} \cdot \alpha_{o_i}) \quad (6.18)$$

where c_{fixed} denotes a stability constant representing the fixed migration overheads (T_{boot} and $T_{overhead}$), and $c_{data} \cdot \alpha_{o_i}$ captures the data transmission phase ($T_{transfer}$). In contrast to the time model, which strictly bounds execution delay, this cost model explicitly limits network bandwidth consumption by imposing heavy mathematical penalties on the migration of state-heavy services.

The battery outage penalty, denoted as $P_{outage}(t)$, represents the risk of an MDC entering a depleted state. This is modelled using an exponential penalty function:

$$P_{outage}(t) = \sum_{m \in M} \exp \left(\lambda \cdot \frac{B_{safe} - B_m(t)}{Capa(EH_m)} \right) \quad (6.19)$$

where λ is a sensitivity coefficient that calibrates the system's aversion to battery outage risks.

An MINLP Formulation

Our goal is to minimise the overall system cost by continuously choosing the best settings of x_{ij} and y_{kl} . By summarising all constraints discussed above, we can formulate the system cost optimisation problem as an MINLP problem:

$$\begin{aligned}
 \mathbf{P1} : & \min : (6.17), \\
 & s.t. (6.9), (6.10), (6.11), (6.14), (6.15) \\
 & x_{ij}, y_{kl}, m_i \in \{0, 1\}, \\
 & \forall o_i \in V, s_j, s_k \in S, f_l \in \mathcal{F}.
 \end{aligned} \tag{6.20}$$

Since the placement decision variable x_{ij} and the frequency scaling variable y_{kl} are integers, whereas the energy harvesting power and incoming request rate are continuous, Problem P1 is formulated as an MINLP problem, which is known to be NP-hard.

Consider a system with m dependent services, n MDC servers, and l discrete CPU frequency levels. The joint configuration space consists of n^m placement combinations and l^n frequency states. Consequently, deriving a global optimum via exhaustive search incurs a computational time complexity of $O(n^m \cdot l^n)$.

For latency-sensitive scenarios with short scheduling intervals (e.g., 100 seconds), standard MINLP solvers (such as Gurobi or Baron) may require hours to converge due to this combinatorial explosion. Thus, finding the global optimal solution is computationally prohibitive for online decision-making. To address this challenge, we propose an effective dynamic resource scheduling algorithm designed for real-time execution.

6.2.8 System Assumptions and Applicability

The real-world applicability of the proposed dynamic scaling and migration strategy is grounded in two primary systemic assumptions. First, the algorithm relies on a Time-Slotted Control paradigm with discrete decision epochs of duration Δt . This assumes that environmental energy fluctuations (e.g., passing cloud cover or sudden wind gusts) and workload spikes occur at a macro-level rather than exhibiting extreme micro-second volatility. Consequently, this model is highly applicable to deployments where environmental changes are relatively smooth, or where the hardware's battery buffers are sufficient to absorb high-frequency noise between epochs.

Second, the DES mechanism adopted in subsequent sections inherently assumes the availability of historical data regarding energy harvesting and task arrivals. While this is highly realistic for established off-grid deployments, such as offshore oil platforms or long-term agricultural IoT networks where weather patterns and user utility are continuously documented, it presents a cold start challenge for rapid-deployment emergency networks like post-disaster recovery infrastructure. In scenarios where historical data is entirely unavailable in the immediate aftermath of deployment, the predictive components of the algorithm would require a burn-in period before achieving optimal scheduling accuracy.

6.3 Algorithm Design

The proposed algorithm consists of two primary components: 1) an initial service assignment and 2) periodic runtime adjustments, which include server frequency scaling and service migration. The workflow of the algorithm is illustrated in Figure 6.3.

Table 6.1: Key notations for the energy management and task scheduling formulations.

Indices	Description
Δt	Constant duration length of each decision epoch t
G	MEC network
M	Set of MDCs
L_s	Number of discrete CPU frequency levels of server s
$\mathcal{F}_s$	Set of available frequency levels of server s
$r(t)$	Volume of request arrivals during the t -th time slot
$E_{demand}(t)$	The aggregate energy demand of MDC m during the t -th time slot
$E_{supply}(t)$	The total energy available of MDC m during the t -th time slot
B_{safe}	Battery safety threshold of energy harvesting device
$B^{init}(t)$	The initial battery level of energy harvesting device
$P_{harvest}$	The harvesting rate of energy harvesting device
$T_{checkpoint}$	The time required for state capturing in service migration
$T_{transfer}$	The time required to move the state data
$T_{restore}$	The restoration time at the destination node
$T_{overhead}$	Constant latency for network reconfiguration
T_{mig}	The total latency of service migration
$\mathbb{I}_{mig}$	Binary indicator function of service migration
$\mathbb{I}_{lb}$	Binary indicator function of service suspension
x_{ij}	Binary variable denoting whether service o_i is placed on server s_j
$m_i^{(t)}$	Binary variable denoting whether service o_i has been migrated
y_{kl}	The frequency state of server s_k with L_s frequency options
C_{mig}	The migration cost
P_{outage}	The battery outage penalty
MDC_{sur}	Group of MDCs with surplus energy
MDC_{def}	Group of MDCs with deficit energy

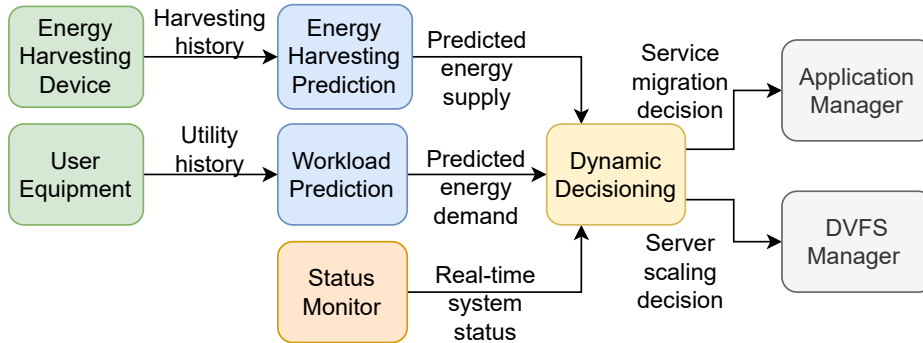


Figure 6.3: The workflow of the dynamic algorithm proposed.

6.3.1 Initial Assignment

The pseudo-code of this initial phase is proposed in Algorithm 4. Guided by the critical path method, this phase proceeds in two main steps. Given an application graph A with its computational demands and dependencies, we first identify the critical path of application, denoted as CP , and then classify service modules as either *critical* or *non-critical*. The services composing CP are assumed to have the largest impact on the end-to-end latency LT . Critical services, represented by V_{Crit} , are prioritised and assigned to MDC servers predicted to yield the lowest latency and energy consumption. Subsequently, non-critical services, denoted by $V_{nonCrit}$, are assigned to MDC servers under the constraint that their placement does not delay the scheduled execution of any successor nodes.

6.3.2 Dynamic Resource Scheduling

During the active runtime phase, our dynamic resource scheduler operates periodically. At the beginning of each time interval, state-of-the-art prediction methods[14] use historical energy harvesting and task arrival records to forecast the energy supply $E_{supply}[m]$, and the volume of new task arrivals for the upcoming period. Based on the current system state and the predicted incoming workload, the future energy demand of each MDC, denoted as $E_{demand}[m]$, is then estimated.

Our focus is not on designing a new forecasting model, but on designing a resilient scheduling mechanism that can tolerate inevitable prediction errors. To enhance the system's robustness against prediction noise, we introduce a pre-processing layer that applies a low-pass filter to the raw data before it is input into the algorithm.

Specifically, we employ a smoothing approach, based on Holt's linear trend method, to smooth out fluctuations in the predicted energy supply. Using this method, we calculate the smoothed level of the predicted energy supply at time t , denoted as $L^{(t)}$, using the following equation:

$$L^{(t)} = \omega_a \cdot E_{predicted}^{(t)} + (1 - \omega_a) \cdot E_{supply}^{(t-1)} \quad (6.21)$$

The smoothing trend, $b^{(t)}$, is updated as follows:

$$b^{(t)} = \omega_b \cdot (L^{(t)} - L^{(t-1)}) + (1 - \omega_b) \cdot b^{(t-1)} \quad (6.22)$$

According to Holt's linear exponential smoothing, we add the smoothing value (L_t) and the smoothing value ($L^{(t)}$) and the smoothing trend value ($b^{(t)}$), to combine the estimated baseline level of the series at time t with its directional trajectory, producing an explicit forecast for future time steps. Consequently, the final smoothed energy supply is expressed as follows:

$$E_{supply} = L^{(t)} + b^{(t)} \quad (6.23)$$

where $\omega_a \in [0, 0.5]$ and $\omega_b \in [0, 0.25]$ represent the level and trend smoothing factors, respectively. A higher ω_a places more weight on the most recent raw prediction, making the model more responsive to sudden changes, while a lower ω_a prioritises the historical data to filter out transient noise and provide a more stable estimate. A similar filtering methodology will also be applied to the predicted incoming request volume.

To facilitate resource management, MDCs are then categorised based on their predicted energy status relative to a safety threshold th , into either a surplus group (MDC_{sur}) or a deficit group (MDC_{def}) as follows:

$$\begin{aligned} E_{supply}[m] &> E_{demand}[m] \cdot th, \forall m \in MDC_{sur} \\ E_{supply}[m] &< E_{demand}[m], \forall m \in MDC_{def} \end{aligned} \quad (6.24)$$

Next, we evaluate the system's overall energy state, categorising it into one of three scenarios, each triggering a distinct policy, as shown in Algorithm 4:

- *Sufficient Scenario*: The entire system is energy-sufficient, meaning all MDCs belong to the

MDC_{sur} group.

- *Deficient Scenario:* The entire system is energy-deficient, meaning all MDCs belong to the MDC_{def} group.
- *Mixed Scenario:* The system contains a combination of energy-surplus and energy-deficient nodes (i.e., both the MDC_{sur} and MDC_{def} groups are non-empty).

Sufficient Scenario Policy

In the Sufficient Scenario, our policy enhances performance via a two-step process, detailed in Algorithm 5:

Server Frequency Scaling-Up: First, the algorithm identifies any active servers operating at sub-maximal CPU frequencies. The server CPU frequencies are then scaled up via DVFS, constrained by the energy budget and the processor's maximum physical frequency. For each server s in an MDC m with surplus energy, we calculate a frequency scaling factor to determine the maximum feasible frequency increase. This factor is derived as follows:

$$scaleUpFactor = \min \left(\frac{f_s^{max}}{f_s^{prev}}, \sqrt{\frac{E_{supply}[m]}{th \cdot E_{demand}[m]}} \right) \quad (6.25)$$

Subsequently, we calculate the new target frequency by multiplying the current frequency f_s^{prev} by this scaling factor. We then apply the floor function so that the result aligns with a supported discrete frequency level:

$$f_s^{target} = \lfloor f_s^{prev} \cdot scaleUpFactor \rfloor \quad (6.26)$$

Critical Service Migration: Next, the policy evaluates potential latency improvements from migrating critical services to less-loaded servers on other MDCs. A potential migration is considered a valid candidate if it both offers a latency reduction and respects the energy budget of the destination. A server s^* is considered a valid migration target for a service v if it satisfies two conditions. First, the

predicted energy state of the target MDC, m , must be in surplus:

$$E_{supply}[m]^* > E_{demand}[m]^* \quad (6.27)$$

Secondly, a migration is only triggered if it yields a net reduction in the predicted end-to-end latency, explicitly accounting for the migration overhead amortised over the projected task volume $r(t)$. This condition is expressed as follows:

$$LT^{target} + \frac{T_{mig}}{r(t)} < LT^{prev} \quad (6.28)$$

Among all valid options, the migration plan offering the greatest latency reduction is selected and denoted as *BestMigTarget*, and the decision is sent to the platform's application manager for execution.

Deficient Scenario Policy

In the Deficient Scenario, where every MDC in the system faces an energy shortfall, the algorithm adopts a conservative, energy-preservation policy, as described in Algorithm 6. The primary objective is to maintain system operation without depleting the limited energy supply. All service migrations are suspended in this scenario. Concurrently, the CPU frequency of all active servers is scaled down to a level that constrains the projected energy consumption within the predicted budget. These decisions are then sent to the system's DVFS manager for execution. For each server s in an energy-deficient MDC m , we calculate a scaling factor to reduce energy consumption to align with the available supply. This reduction factor is derived as follows:

$$scaleDownFactor = \sqrt{\frac{E_{supply}[m]}{E_{demand}[m]}} \quad (6.29)$$

The server's operating frequency is then adjusted by applying this factor and discretising the result to the nearest valid lower frequency level:

$$f_s^{target} = \lfloor f_s^{prev} \cdot scaleDownFactor \rfloor \quad (6.30)$$

Mixed Scenario Policy

The Mixed Scenario is characterised by an energy imbalance across the system, which contains MDCs from both the MDC_{sur} and MDC_{def} groups. The policy aims to resolve this imbalance by reallocating resources from surplus to deficit nodes through a two-phase process detailed in Algorithm 7.

Phase 1. Alleviating Deficits by Migrating Non-Critical Services: To relieve energy-deficient MDCs (MDC_{def}), this phase offloads their non-critical workloads, which have a lower impact on end-to-end latency. First, all non-critical services on these MDCs are identified and sorted by computational demand for sequential migration consideration. The algorithm then sequentially attempts to migrate each service to a suitable server within the surplus group (MDC_{sur}) that has sufficient capacity. After each successful migration, the energy states of the source and destination MDCs are re-evaluated, potentially changing their group status. This process continues until all non-critical services from the original deficit list have been considered.

Phase 2. Handling Critical Services on Remaining Deficit MDCs: After the non-critical migrations, the policy addresses any critical services still located on MDCs that remain energy-deficient. For each of these critical services, two competing actions are evaluated based on the cost gain to determine the optimal outcome:

- Option A (Migration): Find the best possible migration target server within the MDC_{sur} group.
- Option B (Frequency Scaling): Calculate the reduced CPU frequency at which the service's current host server must operate to stay within its local energy budget.

The cost gain is expressed as follows:

$$CostGain = \omega_L \cdot (LT^{prev} - LT^{target} - \frac{T_{mig}}{r(t)}) - \omega_M \cdot C_{mig}(t) \quad (6.31)$$

where the battery outage penalty can be ignored because of the hard energy constraint check.

The algorithm then estimates the resulting cost gain for both options. The action that leads to the higher cost gain is selected and executed (either as a migration command or a DVFS request). The system's state is updated before proceeding to the next critical service on a deficit node.

Algorithm 4: Dynamic Scaling & Migration Algorithm (DSM)

Data: Application $A = (V, E)$, MEC network $G = (M, L)$

Result: Server placement map $P : V \rightarrow S$, scaling and migration decisions

```

1 1: Initial service assignment:
2 Initialise  $P : V \rightarrow S$ , critical path  $CP$ ;
3 for  $v \in CP$  do
4   Find MDC  $m^*$  that minimises  $EST(v)$ ;
5   Select  $s$  in  $m^*.servers$  with min  $EC(v, s)$ ;
6    $P(v) \leftarrow s$ ;
7 end
8  $V_{nonCrit} \leftarrow V \setminus CP$ ;
9 for  $v \in V_{nonCrit}$  do
10  Find MDC  $m^*$  that  $EST(v) + T_{comm}(v, v.succ) \leq EST(v.succ)$ ;
11  Select  $s$  in  $m^*.servers$  with min  $EC(v, s)$ ;
12   $P(v) \leftarrow s$ ;
13 end
14 2: Server scaling and service migration:
15 while  $A.active$  do
16   Estimate  $E_{supply}, E_{demand}$  from predictions;
17   Determine  $MDC_{sur}, MDC_{def}$ ;
18    $M.scenario = Assess(M)$ ;
19   if  $M.scenario == Sufficient$  then
20     Apply the Sufficient Policy;
21   else if  $M.scenario == Deficient$  then
22     Apply the Deficient Policy;
23   else if  $M.scenario == Mixed$  then
24     Apply the Mixed Policy;
25 end

```

Algorithm 5: Sufficient Policy

```

for  $m \in MDC_{sur}$  do
   $E_{sur} \leftarrow E_{supply}[m] - E_{demand}[m];$ 
  for  $s \in m.servers$  do
    if  $s.freq < s.max\_freq$  then
      Calculate  $scaleUpFactor$  with  $m, E_{sur};$ 
       $f_{s_{new}} \leftarrow \lfloor f_s \times scaleUpFactor \rfloor;$ 
    end
  end
for  $v \in CP$  do
   $targetServer \leftarrow \text{BestMigTarget}(v, MDC_{sur});$ 
  if  $targetServer$  then
     $P(v) \leftarrow \text{TargetServer};$ 
     $\text{Migrate}(v, TargetServer);$ 
  end

```

Algorithm 6: Deficit Policy

```

for  $m \in MDC_{def}$  do
   $E_{def} \leftarrow E_{demand}[m] - E_{supply}[m];$ 
  for  $s \in m.servers$  do
    Calculate  $scaleDownFactor$  with  $m, E_{def};$ 
     $f_{s_{new}} \leftarrow \lfloor f_s \times scaleDownFactor \rfloor;$ 
  end
end

```

Algorithm 7: Mixed Policy

```

for  $m \in MDC_{def}$  do
   $V_{migrate} \leftarrow \{v | P(v) = m, v \notin CP\}$ ;
  Sort  $V_{migrate}$  by consumption;
  for  $v \in V_{migrate}$  do
     $tgtServer \leftarrow \text{BestMigTarget}(v, MDC_{sur})$ ;
     $P(v) \leftarrow tgtServer$ ;
    Migrate( $v, TargetServer$ );
  end
end
for  $m \in MDC_{def}$  do
   $V_{critical\_deficit} \leftarrow \{v | P(v) = m, v \in CP\}$ ;
  for  $v \in V_{critical\_deficit}$  do
     $tgtServer \leftarrow \text{BestMigTarget}(v, MDC_{sur})$ ;
    Calculate  $scaleDownFactor$  with  $m, E_{def}$ ;
    Calculate  $LT_{mig}$  and  $LT_{sca}$ ;
    Calculate  $CostGain_{mig}$  and  $CostGain_{sca}$ ;
    if  $CostGain_{mig} > CostGain_{sca}$  then
       $f_{s_{new}} \leftarrow \lfloor f_s \times scaleDownFactor \rfloor$ ;
    else
      Migrate( $v, tgtServer$ );
    end
  end
end

```

6.3.3 Algorithm Complexity Analysis

The time complexity of the proposed robust heuristic algorithm is analysed with respect to the number of services V , MDCs N , and servers per MDC M . The Initialisation phase involves DAG topological analysis and initial service placement. While the DAG analysis operates in linear time relative to dependencies, the placement logic dominates this phase as it iterates through V services to identify optimal resources within the total search space of $N \times M$ servers, yielding a complexity of $\mathcal{O}(V \cdot N \cdot M)$.

In the Dynamic phase, the complexity is operation-dependent. The scaling operation scale linearly with the infrastructure size, $\mathcal{O}(N \cdot M)$, as it inspects server states. The migration operation involves re-evaluating placement for a subset of services; in the worst-case scenario where extensive migration is triggered, this approaches the initialisation complexity of $\mathcal{O}(V \cdot N \cdot M)$.

Consequently, the overall time complexity of the algorithm is derived as $\mathcal{O}(V \cdot N \cdot M)$. This confirms that the proposed approach is a polynomial-time heuristic, yielding computational efficiency suitable for online execution.

The linear relationship with respect to the number of services V , MDCs N , and servers M enables the algorithm to scale gracefully. In real-world terms, as an IoT deployment grows from a small cluster of nodes to a massive, highly distributed network (e.g., expanding an agricultural monitoring grid from 10 to 1,000 sensors), the computational overhead of our scheduler will not become a bottleneck. This polynomial-time complexity enables scheduling decisions to be computed within the strict time limits of the discrete interval Δt , validating its scalability for large-scale, online execution.

6.4 Preliminary experiments

To evaluate the proposed scheduling scheme, we conduct preliminary experiments and a case study. For the preliminary experiments, we assume perfect prediction inputs and seamless migration, where

the only incurred cost is data transfer time.

6.4.1 Performance Indicators & Setup

For our testbed, we used and extended the YAFS fog simulator [65] to support the sequential processing of dependent tasks and dynamic energy harvesting. Network topologies were generated using the Barabasi-Albert model. Each MDC is powered exclusively by an energy harvesting device with battery storage, featuring time-varying harvesting power as shown in Figure 6.4. To prolong the operational lifespan of the battery, a safe line energy threshold is defined at 10%; i.e., when the battery level drops below this threshold, the corresponding MDC becomes non-operational until sufficient energy is harvested and the battery is recharged. Regarding MEC characteristics, the number of MDCs, denoted by n , was varied as $n \in \{2, 4, 8\}$. Similarly, the number of servers, denoted by m , within each MDC was varied as $m \in \{1, 2, 4\}$. The computing resources are heterogeneous, with their CPU frequencies uniformly distributed between 1.5 GHz and 2.5 GHz, representative of low-power edge processing nodes. The propagation delay is set to 5 ms, the bandwidth to 1 Gbps, for standard local edge networks. We also set the device power factor to $10^{-27} \text{ W} \cdot \text{s}^3$, following established literature in [148].

To simulate varying environmental conditions, the ambient energy harvesting rate is modulated by a charging multiplier, denoted as $cm \in \{0.75, 1.0, 1.25, 1.5\}$. This coefficient linearly scales the baseline power generation to reflect discrete states of resource availability, ranging from highly restricted to optimal harvesting conditions. The maximum capacity for each battery was set to 3600 J, and the initial battery level for each MDC was randomly set within the range of 25% to 75% of this maximum. To make our results more general and realistic, we used real-world workloads from the Alibaba cluster trace dataset. This dataset includes information about DAGs from actual workloads running on a large data centre, including services like video streaming and machine learning inference [100]. For our experiments, we randomly chose 10 complex applications from the dataset and present their main

Job id	$ V $	$ E $	Max degree	Average transfer volume	Average workload
1	12	9	3	39.33	9.50
2	16	17	6	29.00	257.88
3	16	17	7	23.63	40.50
4	17	17	5	42.82	13.53
5	16	17	3	46.06	35.75
6	12	11	6	38.67	8.17
7	10	10	4	44.30	14.40
8	10	9	5	35.60	7.10
9	16	16	2	47.19	1.00
10	10	9	4	43.40	32.70

Table 6.2: Application characteristics.

characteristics in Table 6.2.

The columns in Table 6.2 define the structural and computational parameters of the evaluated application workflows: $|V|$ denotes the total number of microservice component modules within the DAG; $|E|$ denotes the number of dependency data-flow edges; Max degree represents the highest number of connected edges incident to any single module; The average transfer volume corresponds to the mean size of inter-service messages ($size(pkt)$), measured in Megabytes (MB), whereas the average computational workload corresponds to the mean processing demand per task instance ($wl(o)$), measured in Megacycles (10^6 CPU cycles).

To realistically model ambient renewable energy generation, real-time service requests, and the temporal mismatch between energy harvesting and user utility, we adopt wave-based synthetic traces for both generation and incoming traffic. As illustrated in Figure 6.4, task arrivals at the MDCs and harvested energy both exhibit characteristic peaks and valleys, with energy generation reaching a peak rate of approximately 2 kW to simulate small-scale renewable installations.

To evaluate our proposed strategy, we manage to identify suitable external baselines. Our work addresses a specific, multi-faceted problem that combines 1) DAG-based task dependencies, 2) exclu-

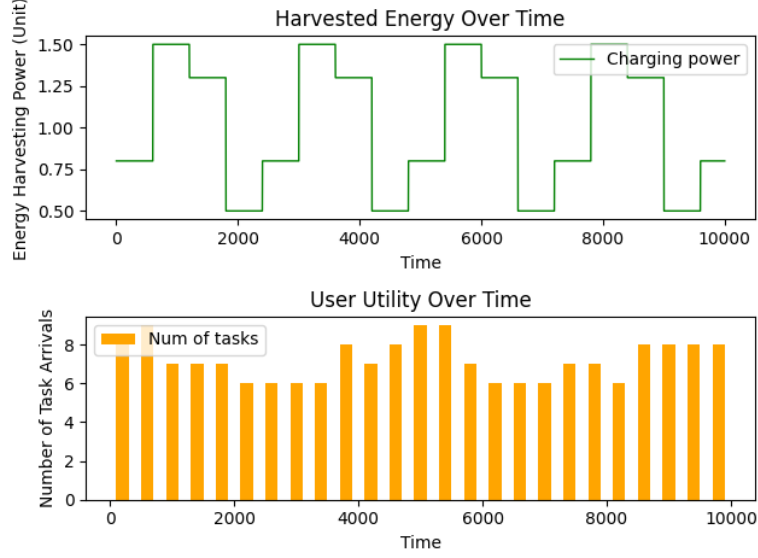


Figure 6.4: The time-varying energy harvesting power and task arrival rate used to simulate the temporal mismatch between energy supply and user demand.

sive reliance on harvested energy, and 3) the joint, online optimisation of both DVFS and service migration. While existing research addresses subsets of these challenges, we found no contemporary schedulers that holistically integrate all of these components. Adapting algorithms from these related but distinct domains would require significant modifications, potentially leading to an unfair comparison.

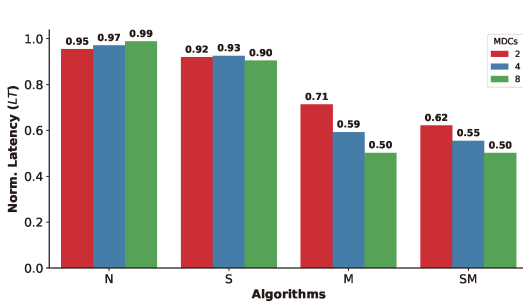
To clearly isolate and quantify the specific contributions of each component of our strategy, we adopt an ablation study approach, and compare four different scheduling policies. The first policy, denoted as N , takes no adaptive action after the initial placement of the services. The second policy, S , only applies dynamic scaling of server frequencies. The third policy, M , selectively migrates specific services to servers within the same or different MDCs. Finally, the SM policy combines both scaling and migration strategies. The initial service placement on MDCs is performed using the EDEM algorithm [25, 26]. This approach allows for a clear and unambiguous analysis of how each adaptive strategy contributes to system performance under the stringent constraints of a purely energy-harvesting MEC environment.

Each experimental configuration was simulated over 10,000 global time steps and repeated 5 times to obtain reliable average results. Scheduling decisions are made every 100 global time steps to adapt to the continuous arrival of new data from IoT sensors. The simulations were conducted on a server with a 4x Intel Xeon Gold 6230N CPU, 256 GB of RAM, and the Ubuntu 20.04 operating system.

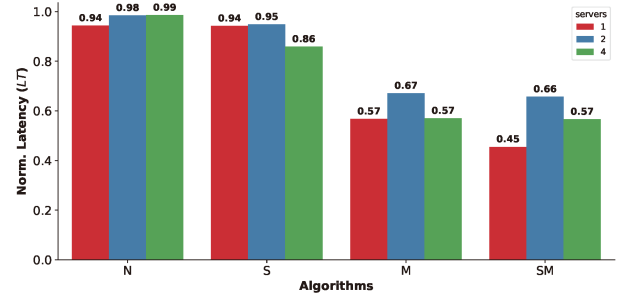
We evaluate the performance of the algorithms using the following three metrics:

- Average User-Experienced Latency (LT): the mean response time for processed user requests, extracted from message timestamps stored in the simulation traces.
- Throughput (TH): the total number of requests successfully executed.
- Resource Utilisation (RU), a measure of how effectively the available resources are used. A total of 7200 experiments were conducted.

6.4.2 Ablation Study Results



(a) Varying number of MDCs ($n \in \{2, 4, 8\}$).



(b) Varying number of servers per MDC ($m \in \{1, 2, 4\}$).

Figure 6.5: Impact of scaling the number of MDCs and servers on normalised latency (LT).

The results were first averaged across runs, and then LT was normalised per experimental configuration (i.e., for each unique combination of application, number of MDCs, servers, and charging multiplier). This normalisation was used since directly comparing raw results can be misleading. For LT , each latency value was divided by the maximum latency observed within its respective config-

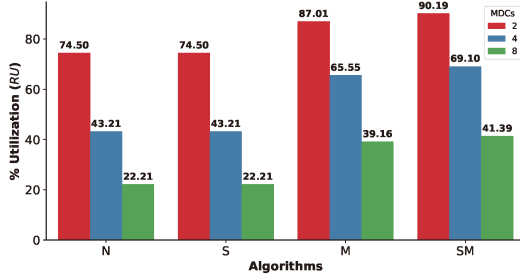
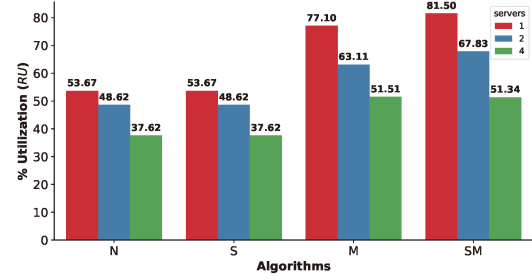
(a) Varying number of MDCs ($n \in \{2, 4, 8\}$).(b) Varying number of servers per MDC ($m \in \{1, 2, 4\}$).

Figure 6.6: Impact of scaling the number of MDCs and servers on resource utilisation (RU).

uration group. Under this normalisation, the worst-performing algorithm receives a score of 1, and better-performing ones receive proportionally smaller values.

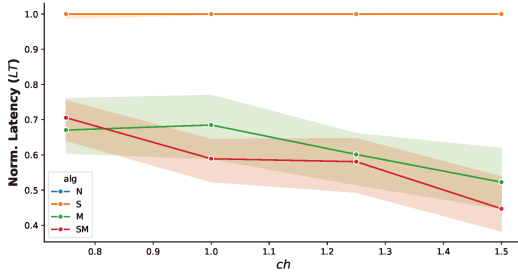
In Figure 6.5a and Figure 6.5b, we plot the normalised latency for all algorithms as the number of MDCs/servers increases. We observe that the N algorithm shows high latency across all settings. The S algorithm performs marginally better than N , but its performance indicates that scaling alone is not suitable for latency-sensitive systems. On the other hand, the Migration-only policy (M) and the combined SM policy are consistently the best-performing algorithms in terms of latency. The M policy improves latency significantly as the number of MDCs increases. Both M and SM show strong improvements when only one server is available, but this improvement does not scale proportionally for two and four servers.

Specifically, when analysing server scaling in Figure 6.5b, we observe a non-linear latency trend for the combined SM policy across server configurations ($m \in \{1, 2, 4\}$). Note that lower normalised values represent superior latency performance. The single-server configuration ($m = 1$) achieves the lowest normalised latency (0.45), whereas $m = 2$ exhibits a latency peak (0.66), which subsequently improves to 0.57 at $m = 4$.

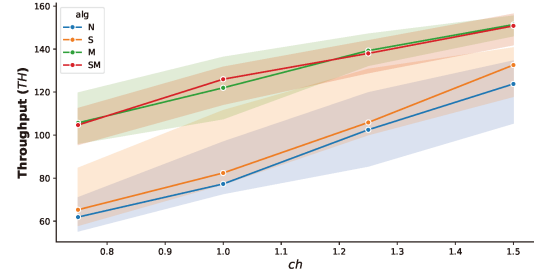
This non-linear behaviour stems from a trade-off between host co-location benefits, inter-server state migration overheads, and local compute headroom. When $m = 1$, all services allocated to an MDC

are co-located on a single physical server, completely avoiding inter-server network transmission and checkpointing delays. Transitioning to $m = 2$ enables the *SM* algorithm to perform intra-MDC service migrations; however, fragmenting tightly coupled services across two physical hosts introduces inter-process communication barriers and migration overheads (T_{mig}). Because two servers provide limited parallel capacity, local queuing contention during energy-deficit intervals further exacerbates latency. Conversely, scaling to $m = 4$ servers per MDC provides sufficient parallel processing headroom and low-load target servers within the local edge site, allowing the *SM* scheduler to offload tasks seamlessly without severe queuing bottlenecks, thereby reducing normalised latency to 0.57.

In Figure 6.6a and Figure 6.6b, we plot the Resource Utilisation (*RU*) for all algorithms as the number of MDCs/servers increases. As a general trend, we observe that as resources become more distributed, *RU* decreases significantly for all algorithms. *SM* maintains the highest *RU* across all settings due to its dynamic scaling and migration features. The Scaling-only algorithm *S* fails to offer a significant improvement compared to the baseline (*N*).



(a) Normalised latency vs. charging multiplier.

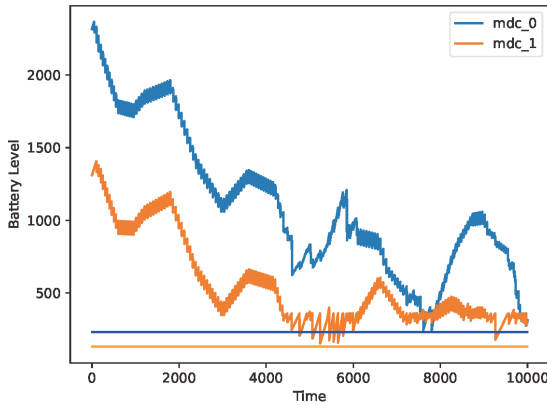


(b) Throughput vs. charging multiplier.

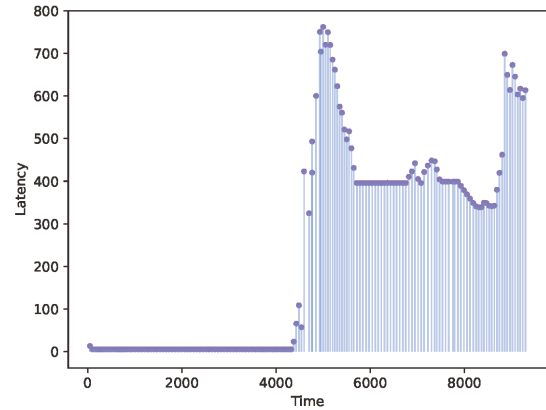
Figure 6.7: Impact of the energy harvesting charging multiplier (ch) on normalised latency (LT) and throughput (TH).

In Figure 6.7a, we demonstrate the normalised latency for all four algorithms as a function of the charging multiplier (ch). The *N* and *S* algorithms consistently exhibit the worst latency regardless of the charging rates. For low ch values, *M* slightly outperforms *SM*; however, as ch increases, *SM* proves more adaptive to energy changes. This behaviour suggests that *SM* avoids overloaded resources by

using its migration capabilities. Similarly, in Figure 6.7b, we analyse the changes in throughput (TH) as ch increases. We observe that the Migration-only (M) policy maintains a high throughput rate and that the combined SM policy does not significantly increase throughput over M alone. Increasing ch up to 1.5 does not appear promising, as the resulting increase in throughput (TH) becomes marginal. To conclude the experimental evaluation, this section presents a detailed example of the SM algorithm's operation. For fairness and to better illustrate its behaviour, we intentionally selected a case where SM exhibits moderate performance. In this experiment, two MDCs are used, each equipped with 4 servers and a charging multiplier of 1.25. Figure 6.8a shows the battery levels (i.e., remaining capacity) of both MDCs over time, with horizontal red lines indicating their respective safety thresholds.



(a) Battery levels of both MDCs over time.



(b) Distribution of user-experienced latency over time.

Figure 6.8: Detailed performance of the SM algorithm in a case study with 2 MDCs and 4 servers per MDC ($ch = 1.25$).

Initially, both MDCs are actively utilised by the initial placement algorithm (EDEM). During the first 4000 time steps, latency remains very low (as shown in Figure 6.8b) because both MDCs have sufficient energy to process incoming tasks. However, a critical period arises between time steps 4000 and 6000, during which MDC 1 runs out of energy. According to Figure 6.4, the energy harvesting

capability during this interval is nearly zero, while the task arrival rate increases, causing the energy bottleneck.

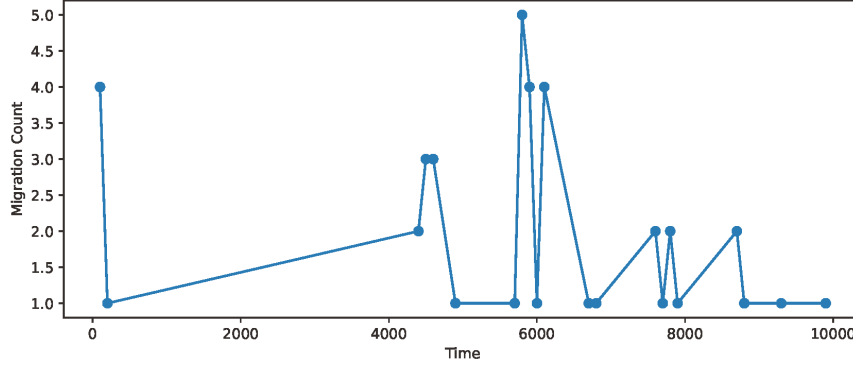


Figure 6.9: Timeline of service migrations in the case study, showing most migrations occur during the low-energy period between 4000 and 8000 time steps.

As depicted in Figure 6.9, most service migrations occur during this low-energy phase, relocating services to MDC 0. This increased load causes MDC 0 to deplete its own battery around time step 8000. Meanwhile, MDC 1 has recovered some energy, prompting the scheduler to migrate services back to it, thereby helping reduce latency once again. Towards the end of the simulation, the demand for processing exceeds the system's charging capabilities, resulting in a slight increase in overall latency. Finally, the heatmap in Figure 6.10 demonstrates that the migration policy is effective and well-balanced, as it avoids any "ping-pong" behaviour.

6.5 Extended Experiments

While the preliminary experiments demonstrated the effectiveness of our approach, we further evaluate the algorithm under more realistic scenarios to assess its robustness. Instead of assuming perfect, oracle-like predictions for the algorithm's inputs, we simulate dynamic prediction errors by introducing Gaussian noise. Additionally, we examine different migration scenarios encompassing both low and high migration costs.

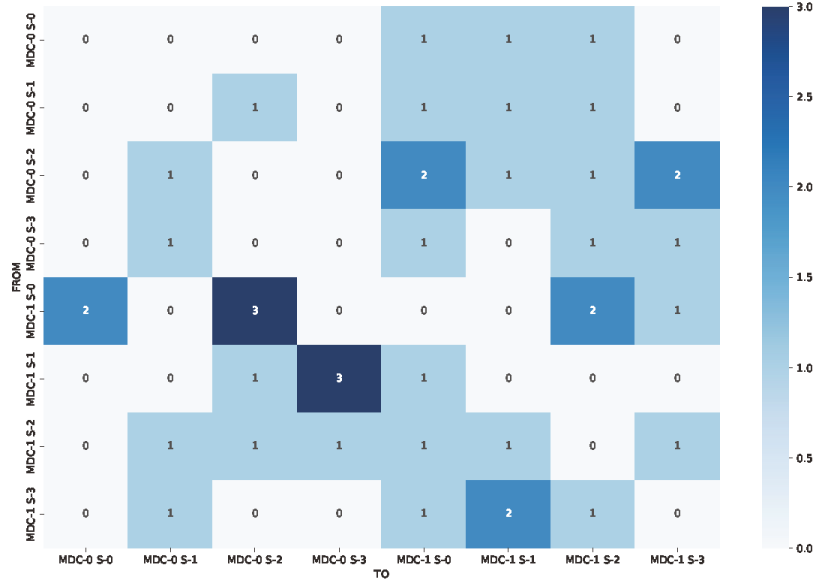


Figure 6.10: Heatmap of service migration counts between servers in the case study, showing a balanced distribution and no "ping-pong" effects.

The energy harvesting and user utility simulated in these extended experiments follow the model depicted in Figure 6.11. Compared to the previous wave-based model, we introduced greater dynamism to better reflect real-world conditions.

We extend our experiments to analyse the impact of the following parameters:

- DES smoothing factors
- Migration weight
- Macro migration time scaling factor
- Battery safety margin
- Prediction error

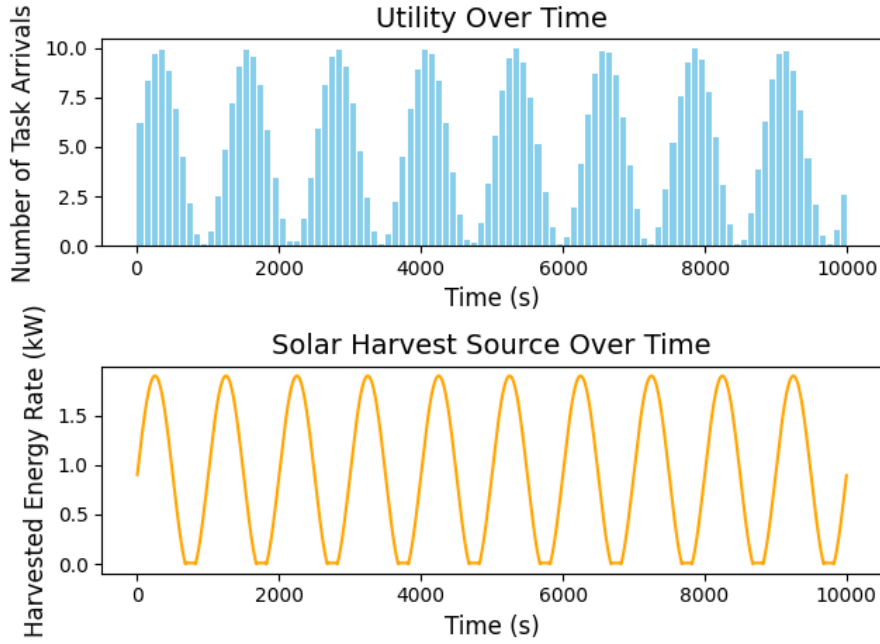


Figure 6.11: The dynamic energy harvesting and task arrival model used for the extended experiments, reflecting more realistic, fluctuating conditions.

6.5.1 Impact of DES Smoothing Factors

We first analyse the impact of the DES smoothing factors, ω_a and ω_b . Since DES serves as the pre-processing stage for our proposed algorithm, we aim to investigate its performance across varying levels of prediction noise and select the optimal parameter pair for subsequent experiments.

We conducted experiments using 4 MDCs and 16 servers under varying prediction noise levels. We tested the following (ω_a, ω_b) pairs: (0.2, 0.05), (0.3, 0.1), (0.4, 0.15), (0.4, 0.1), (0.5, 0.1), and (0.5, 0.2), alongside (1, 0), which serves as the non-smoothing baseline. After normalising the results across multiple runs, we compared three main metrics: average end-to-end latency, task processing throughput, and average energy consumption per task.

As shown in Figures 6.12 and 6.13, as the standard deviation of the generated prediction noise grows from 0.1 to 0.5, the performance gap between the baseline and the smoothed configurations widens. Smoothing factors with ω_a set to 0.4 and 0.5 consistently yield lower latencies and exhibit signifi-

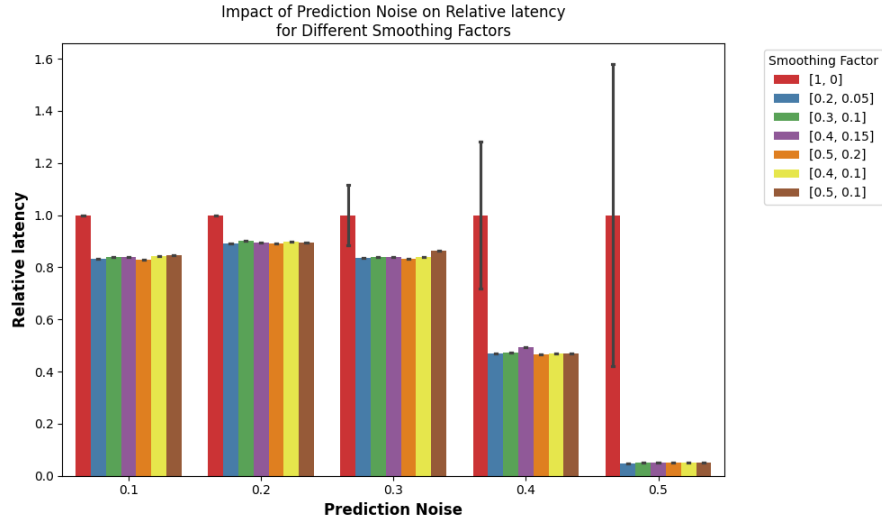


Figure 6.12: Impact of different DES smoothing factor pairs (ω_a, ω_b) on average latency under increasing prediction noise levels.

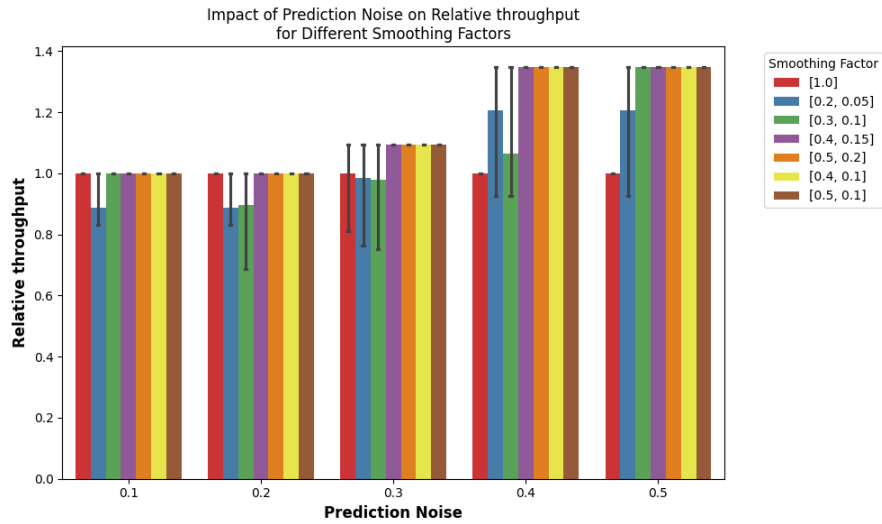


Figure 6.13: Impact of different DES smoothing factor pairs (ω_a, ω_b) on task throughput under increasing prediction noise levels.

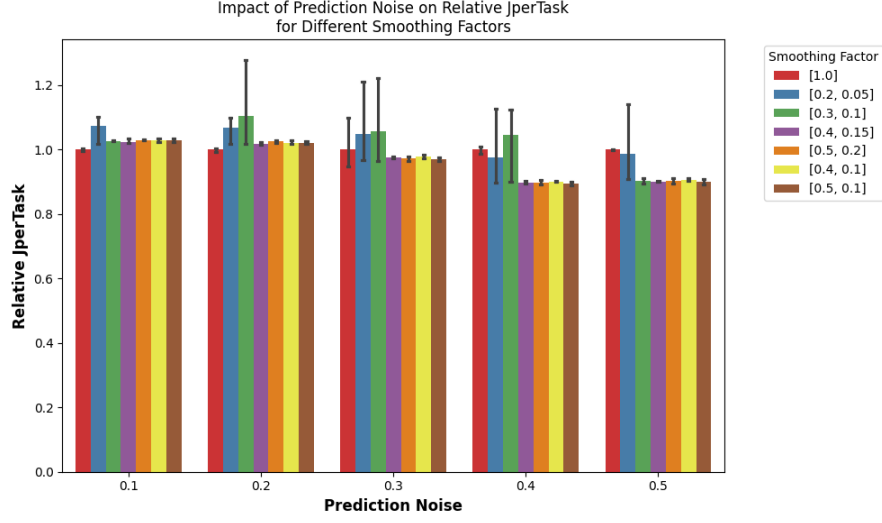


Figure 6.14: Impact of different DES smoothing factor pairs (ω_a, ω_b) on average energy consumption per task under increasing prediction noise levels.

cantly higher processing throughput as the noise increases. Similarly, as shown in Figure 6.14, the average energy consumption for task processing in these configurations remained slightly lower than the baseline.

From these results, we conclude that smoothing factors with an ω_a of 0.4 or 0.5 demonstrate superior stability when mitigating prediction noise, while remaining adaptive to the changing environment. Therefore, we select (0.4, 0.15) as the fixed values for (ω_a, ω_b) to be used in the subsequent impact analyses.

6.5.2 Impact of Migration Weight

The migration weight, denoted as ω_M , determines how the algorithm calculates the cost-benefit trade-off of scaling and migration options, thereby influencing its scheduling tendencies. Keeping the prediction noise, battery safety margin, and migration overhead scaling factor at their minimum values, we analyse the impact of the migration weight by evaluating the migration count and average latency as ω_M varies from 0.1 to 4.0.

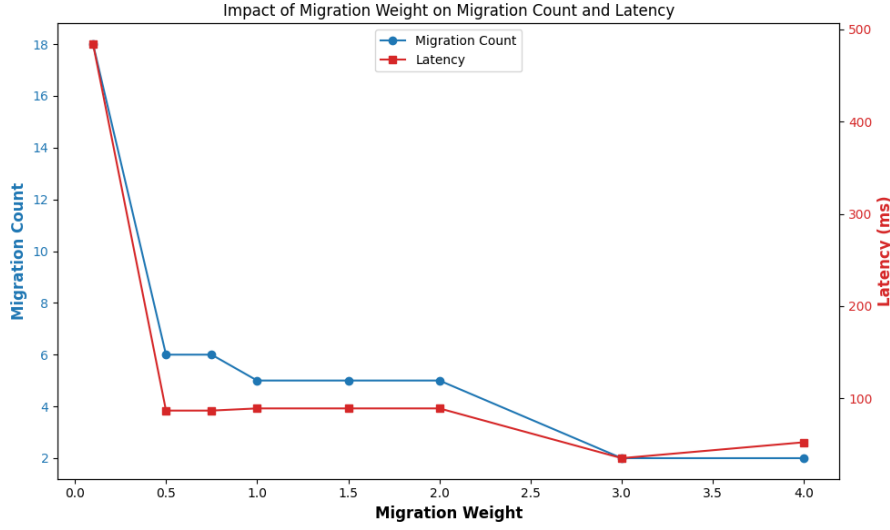


Figure 6.15: Impact of the migration weight (ω_M) on the total number of migrations and the resulting average latency.

As shown in Figure 6.15, when $\omega_M = 0.1$, the perceived system cost of migration during the decision-making process is negligible. Consequently, service migrations are triggered excessively, incurring substantial real-world overhead, which is reflected in the high average latency. As ω_M gradually increases, the migration strategy becomes more conservative; the reduction in the number of migrations leads to improved latency. This positive trend continues until ω_M exceeds 3.0. Beyond this threshold, the algorithm penalises migrations too heavily, resulting in significantly fewer migrations overall. The scheduler tends to delay necessary migrations, rendering the system less adaptive to environmental changes and causing latency to rise again. With a sharp reduction in migration frequency and only a marginal increase in latency during this optimal transition, the results indicate that our algorithm successfully eliminates non-beneficial, "ping-pong" migrations.

6.5.3 Impact of Migration Overhead

To simulate varying environmental conditions and their associated migration overheads, we applied a multiplier, ω_O , as a macro migration scaling factor to the migration time and investigated its impact on the algorithm. We compared the total number of migrations scheduled by our dynamic scaling-

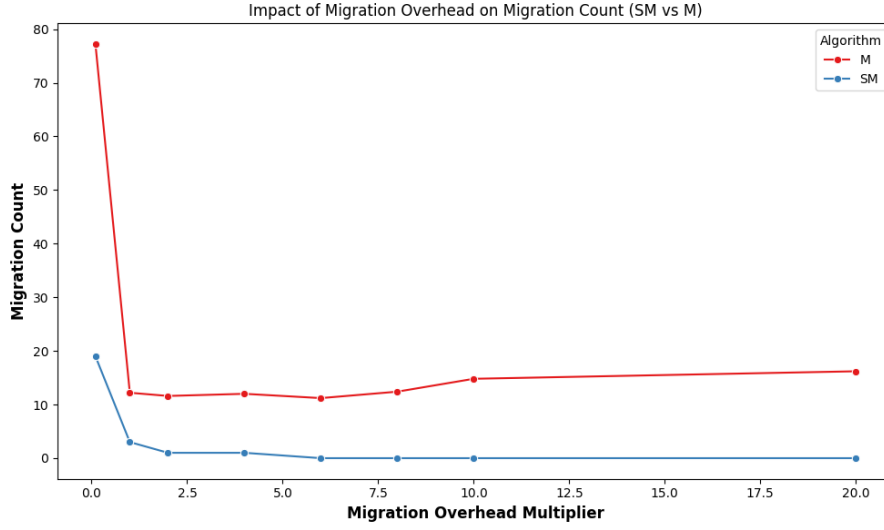


Figure 6.16: Impact of the migration overhead scaling factor (ω_O) on the total number of migrations for the proposed policy versus a migration-only baseline.

and-offloading policy against a migration-only baseline, with ω_O ranging from 0.1 to 20.0.

As shown in Figure 6.16, when increasing migration overhead indicates degrading network conditions, our proposed policy autonomously becomes more conservative, performing fewer migrations to mitigate network congestion. In contrast, the migration-only policy remains aggressive and exhibits less sensitivity to deteriorating environmental conditions. These results validate the adaptability of our algorithm to dynamic network environments and its effectiveness in minimising unnecessary system overhead.

6.5.4 Impact of Battery Safety Margin

The battery safety margin, denoted as B_{safe} , serves as the minimum acceptable battery level required for the energy-harvesting-powered devices to operate safely. To mitigate the risk of battery depletion, we integrated this threshold into our algorithm to enhance energy reliability. We varied the battery safety margin from 0.001 to 0.30 and monitored the resulting changes in renewable energy utilisation and system throughput.

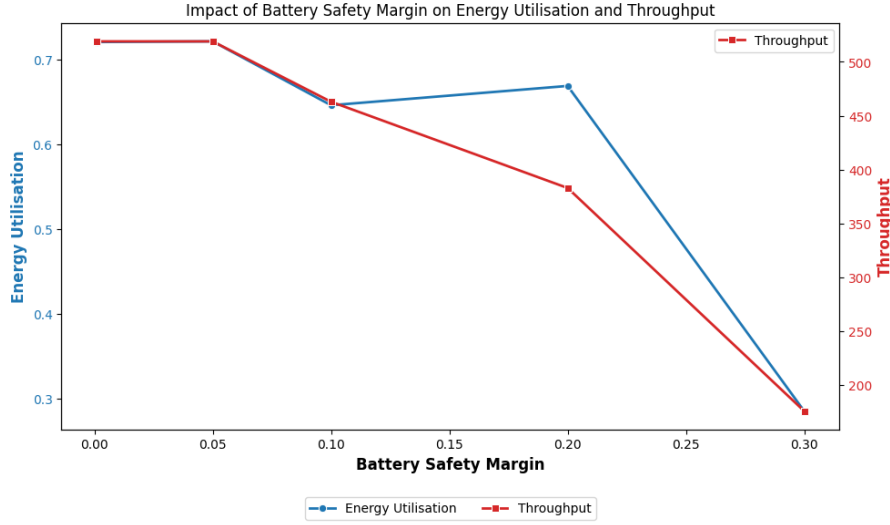


Figure 6.17: Impact of the battery safety margin (B_{safe}) on renewable energy utilisation and system throughput.

Energy utilisation, calculated as the ratio of server energy consumption to the total harvested energy, remains high when the battery safety margin is between 0.001 and 0.05, but gradually declines as the margin increases, as shown in Figure 6.17. A similar trend is observed for system throughput. By migrating services or scaling down local servers if the algorithm predicts that the battery level will drop below the safety margin in the next time slice, our strictly conservative energy policy carefully avoids battery depletion. Consequently, both evaluation metrics demonstrate strong performance even when the battery safety margin starts very low. However, the policy fails to fully utilise the harvested energy when the margin exceeds 0.15. In such cases, the algorithm becomes overly cautious, triggering unnecessary migrations and scaling operations even when the battery level is practically sufficient.

Therefore, a safety margin in the range of 0.001 to 0.05 is identified as the optimal operating point, maximising throughput while maintaining a near-zero risk of power outage.

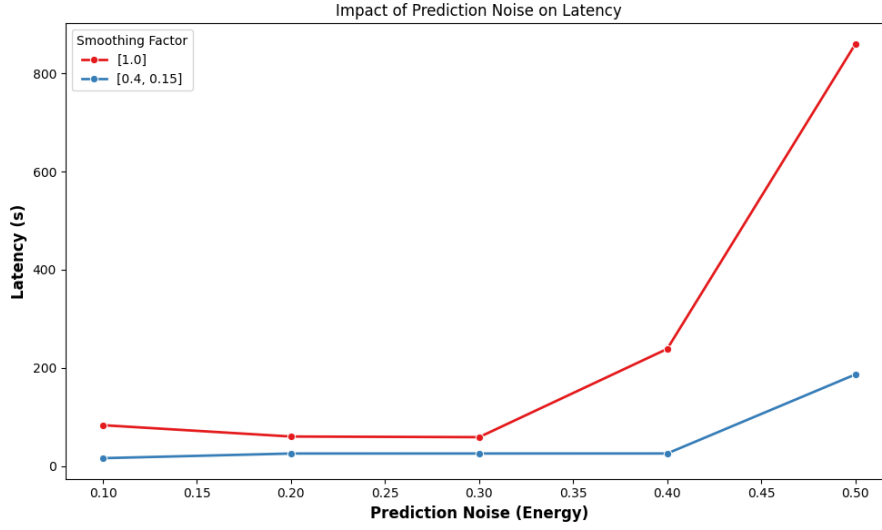


Figure 6.18: Robustness comparison of the algorithm with smoothed vs. non-smoothed inputs under increasing prediction error.

6.5.5 Impact of Prediction Error

Prediction error is a critical factor influencing the precision of scheduling decisions. The algorithm's ability to respond to varying levels of prediction error serves as a key indicator of its robustness. We varied the simulated prediction noise from 0.1 to 0.5 and compared the overall latency using smoothed inputs against a non-smoothed baseline.

As shown in Figure 6.18, as the standard deviation of the generated Gaussian noise increases from 0.1 to 0.3, the resulting latency of our algorithm with smoothed inputs remains at a consistently low level, distinctly outperforming the non-smoothed baseline. As the prediction error continues to increase, the latency for both approaches rises; however, our smoothed approach maintains a significantly lower latency. While the baseline performance exhibits a sharp degradation as the prediction error grows, our algorithm demonstrates a much more graceful degradation. These results validate the robustness of the overall system design, which successfully integrates safety-threshold-based risk prevention with data-smoothing-based input filtering.

6.5.6 Summary

The extended experiments validate the robustness and adaptability of the proposed dynamic scaling-and-offloading algorithm under realistic, noisy edge computing conditions.

By introducing dynamic prediction errors and varying network overheads, the evaluation demonstrates that the system gracefully degrades and maintains high end-to-end performance compared to non-smoothed and migration-only baselines. The integration of data-smoothing input filtering effectively mitigates prediction noise, while calibrated safety margins and migration weights allow the scheduler to autonomously eliminate inefficient "ping-pong" migrations, avoid network congestion, and maximise renewable energy utilisation. Ultimately, the algorithm successfully balances task scheduling, energy reliability, and network overhead, confirming its viability and efficiency for deployment in unpredictable, real-world environments.

6.5.7 Discussion and Limitations

The extended evaluations confirm that the proposed strategy excels in off-grid environments characterised by distinct temporal and spatial harvest-utility mismatches. By relying on horizontal edge-to-edge migration rather than vertical cloud offloading, the system successfully sustains continuous operation even when localised energy harvesting drops to zero, provided that neighbouring nodes possess surplus energy.

However, the reliance on service migration introduces an inherent system trade-off, governed by the migration weight parameter ω_M . A highly aggressive migration policy (a low ω_M) effectively preserves local battery life but risks saturating the limited edge network bandwidth with state-transfer overheads. Conversely, a conservative policy (a high ω_M) minimises network congestion but forces aggressive local CPU frequency scaling, which inevitably degrades the end-to-end latency for the user. The empirical optimal range identified in our evaluations balances this trade-off, though the exact optimal tuning remains highly dependent on the specific network topology and bandwidth capacity of

the physical deployment.

Furthermore, a fundamental limitation of this purely horizontal, off-grid approach is its vulnerability to prolonged, network-wide energy droughts. In the event of a catastrophic weather phenomenon, such as a multi-day severe storm that simultaneously reduces the charging multiplier across all geographically adjacent MDCs, the system loses the ability to offload tasks to energy-rich neighbours. Under such extreme, absolute worst-case conditions, horizontal migration ceases to be a viable safety mechanism. Unlike hybrid systems that can fall back on the power grid, this stand-alone system would be forced to indiscriminately drop incoming non-critical requests to maintain the physical safety thresholds of the edge batteries.

6.6 Conclusion

In this chapter, we introduced a novel and robust online strategy for MEC systems powered exclusively by renewable energy. Our approach dynamically manages service migration and server frequency scaling to efficiently utilise harvested energy while achieving low service latency and high system throughput. Designed for rapid execution, the algorithm is specifically tailored for noise resilience and environmental adaptability, incorporating conservative mechanisms to mitigate critical risks such as power depletion.

Preliminary experimental results using real-world workflow datasets demonstrated the effectiveness of our algorithm across various operational scenarios. Furthermore, extended experiments incorporating realistic energy and utility models, simulated prediction errors, and varying migration overheads validated the robustness of the algorithm in dynamic environments subject to input noise.

While these results are promising, we acknowledge several limitations that provide clear directions for future research. Although we conducted an ablation study and extensive impact analyses of the proposed algorithm, there remain avenues for refinement. In future work, we plan to evaluate the

algorithm's performance in conjunction with state-of-the-art prediction methods and real dynamic energy and utility data from existing datasets. Additionally, we aim to benchmark our approach against contemporary solutions, particularly lightweight, learning-based algorithms.

Chapter 7

YAFS-MEC: A Simulation Framework for Energy-Constrained Edge Orchestration

In this chapter, we detail the fundamental architectural extensions applied to the YAFS simulator. While the baseline framework provides adequate primitives for generic discrete-event network simulation, it is structurally deficient for modelling dependent service orchestration within energy-constrained MEC environments. The subsequent sections outline the custom modules, execution logic, and stateful energy telemetry engineered to enforce the specific topological, computational, and power constraints required by our evaluation methodology.

7.1 The Standard YAFS Framework

YAFS is an event-based simulator designed for fog computing environments. As shown in Figure 7.1, YAFS comprises several core classes and APIs that expose control over application logic, topology configuration, placement algorithms, service deployment, and task execution. The simulation engine is built on the SimPy library to manage all transmission and computation events.

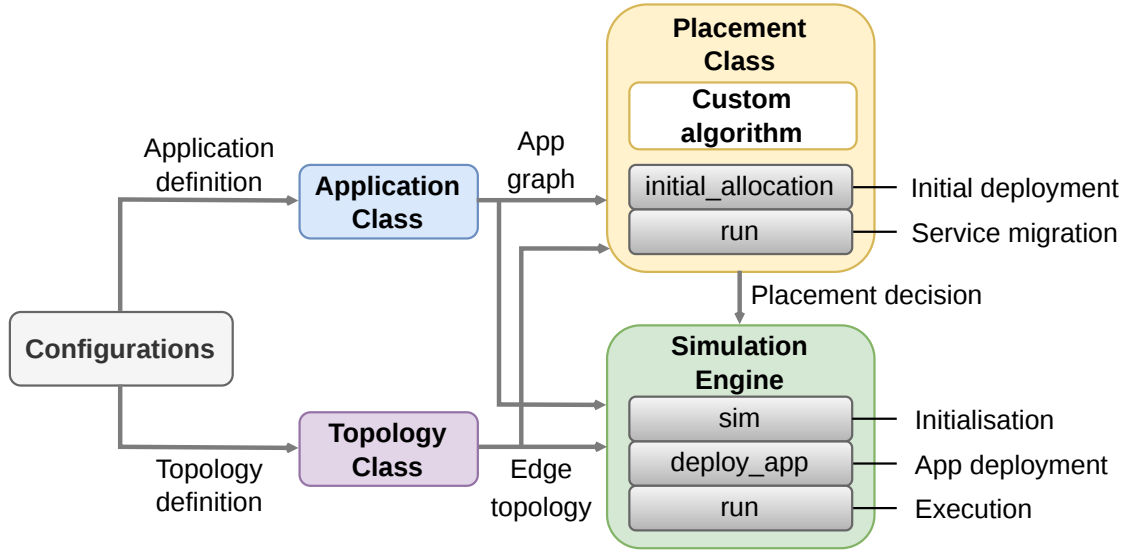


Figure 7.1: Core architecture and execution workflow of the YAFS framework.

As introduced in Section 3.4, the network topology is represented as a graph, featuring customisable connection and machine attributes.

The simulator natively supports applications modelled as DAGs. The entities comprising the DAG, including data sources, services, and end users, are all instantiated from a base **module** class. Dependencies between these modules are defined via **messages**, with each dependency edge in the DAG corresponding to a distinct message type. Service requests act as source messages routed to the DAG's initial services. Request timing is entirely configurable, allowing for one-off transmissions, fixed-interval streaming, or interval timing governed by a specific probability distribution.

Based on message attributes, the simulator routes messages from source to destination along the computed shortest path. A global network process manages system messages during simulation, as illustrated in Figure 7.2. The system network pipe operates as a global bus handling all network traffic. The system continuously reads messages from this pipe, checks their next waypoints, calculates transmission times for the next hop, and initiates corresponding timeout event processes. The SimPy core manages these events via its internal event queue, updating the global virtual time. If a message's

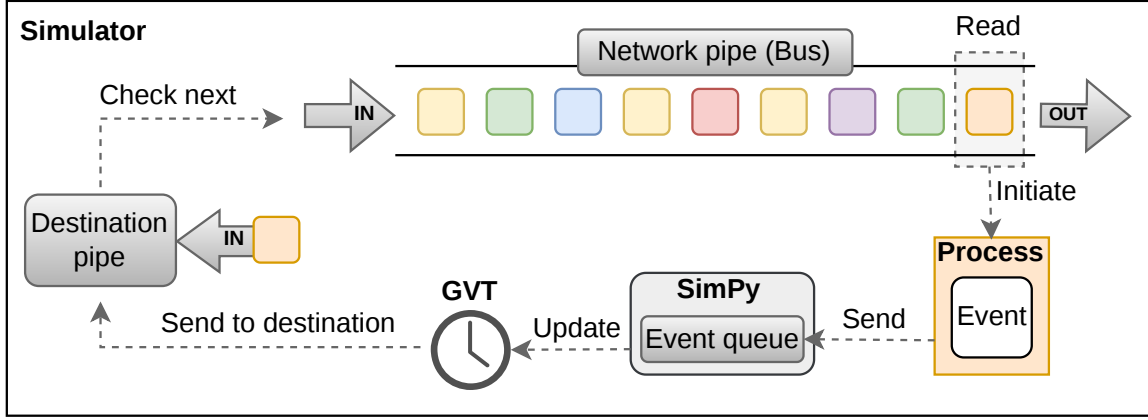


Figure 7.2: Transmission of messages in the YAFS simulator.

next hop is its destination, it is dispatched to the destination module’s consumer pipe; otherwise, it returns to the network pipe for further routing.

This consumer pipe functions as a First-In-First-Out queue dedicated to each instantiated service, buffering all incoming messages. As illustrated in Figure 7.3, the operational lifecycle of a service revolves around this structure. The service continuously polls its local consumer pipe; upon retrieving a message, it executes the designated task, generates the subsequent result messages, and forwards them to successor services within the DAG.

7.2 Architectural Hierarchy and Topology

While YAFS provides fundamental primitives for constructing graph-based network topologies, where nodes represent generic entities and edges represent communication links, it lacks the native architectural hierarchy required to accurately model MEC infrastructures. To simulate dependent services within a realistic MEC environment, fundamental architectural extensions are necessary.

To address this limitation, we extend the core simulator by introducing a hierarchical MEC network model. This extension explicitly defines domain-specific entities, including MECs equipped with

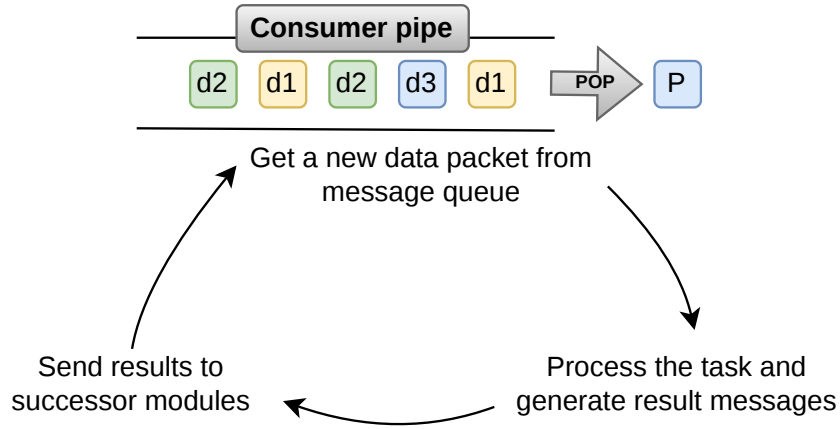


Figure 7.3: The operational lifecycle of a service in the YAFS simulator.

edge servers, UE, and energy-harvesting devices. By bridging our engineered classes with the existing YAFS primitives, we introduce the hierarchical distinctions inherent to the MEC paradigm. This abstraction streamlines topological configuration by acting as a high-level API; rather than manually defining and linking discrete simulation entities, researchers can instantiate MEC environments using our specialised classes, which the framework seamlessly translates into the low-level graph structures required by the simulation core.

Similar architectural refinements were applied to the application modelling layer. We encapsulate the standard DAG service configurations into higher-level 'application nodes'. This abstraction decouples resource orchestration logic from the simulator's underlying mechanics. Consequently, placement algorithms do not need to interface directly with the base YAFS classes; they can simply ingest our streamlined MEC and application models to generate an intuitive and easily visualisable placement plan.

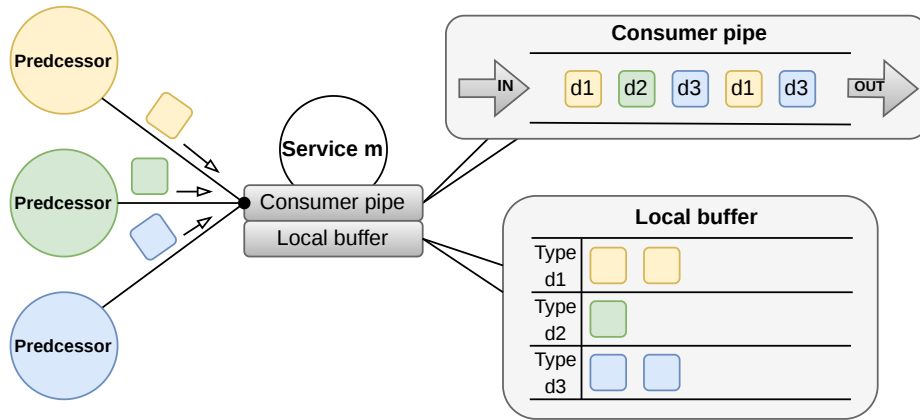


Figure 7.4: Internal structure of a service featuring an extended local buffer.

7.3 Service Dependency and Synchronisation

The default YAFS framework employs an asynchronous trigger mechanism, activating a service upon receiving any single input from its predecessors. However, complex DAG applications enforce strict precedence constraints: a service with multiple incoming directed edges requires the state, payload, or execution completion of all its predecessors before proceeding. To address this, we enforce a strict conjunctive (AND-logic) synchronisation barrier. As depicted in Figure 7.4, we introduce a dedicated message buffer alongside the existing consumer pipe for each service. For a given service m with multiple predecessors, incoming messages are isolated and queued within this local buffer based on their origin.

At the simulation engine level, we fundamentally altered the service lifecycle (Figure 7.5). Messages arriving at the consumer pipe are fetched and allocated to the local buffer categorised by their sender identifier. The service continuously polls the buffer for a complete, synchronised set of prerequisite messages. Upon detecting a complete set, the messages are dequeued, and task execution initiates. Otherwise, the service remains in a blocked state, suspending execution until all topological predecessors have delivered their required payloads.

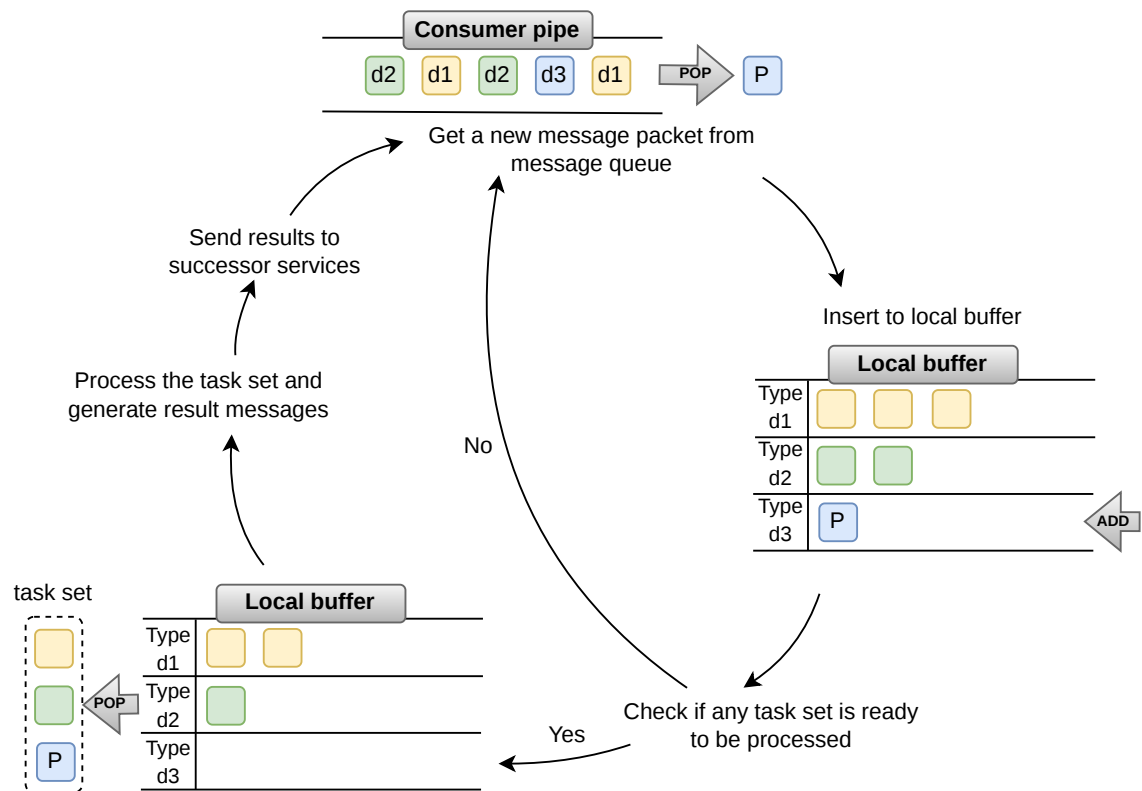


Figure 7.5: The operational lifecycle of a service featuring dependency synchronisation.

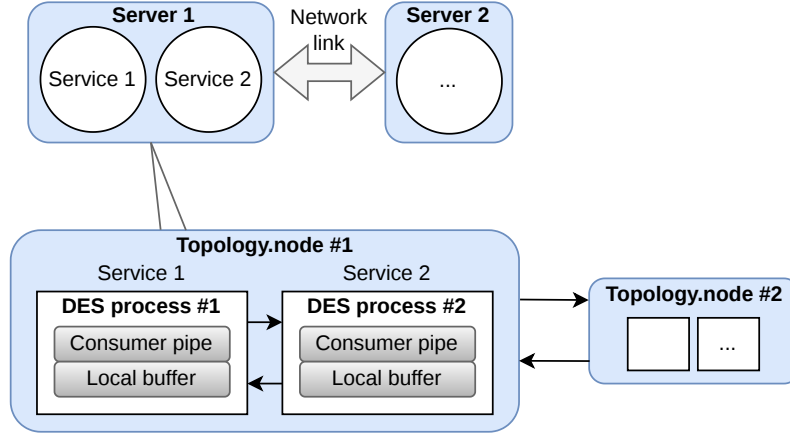


Figure 7.6: Services operate as independent processes in the YAFS framework.

7.4 Resource Contention and Concurrent Task Processing

Within the YAFS framework, each service operates as an independent discrete-event simulation process that governs its own lifecycle. Every node in the network topology is assigned a unique topological address. The placement of a service is, therefore, fundamentally the mapping of a service's process identifier to a specific node's address. Consequently, inter-service communication across the simulated network is executed natively as inter-process message passing, as shown in Figure 7.6.

However, the baseline simulator exhibits a critical limitation regarding resource contention during concurrent task processing. To simulate a processing event, the engine calculates the required execution time and initiates a corresponding timeout event. Natively, this task duration is calculated simply by dividing the task's workload by the total computational capacity of the host server. This calculation implicitly assumes that each task has dedicated access to the server's entire resource pool, completely ignoring the presence of concurrent processes. When multiple services are co-located on a single edge server, this assumption violates physical compute constraints, leading to unbounded resource oversubscription and yielding highly inaccurate performance metrics.

To rectify this architectural limitation, we extended the simulation engine to enforce strict compute

resource pre-allocation for every deployed service. Task execution durations are now calculated based exclusively on the specific quantum of resources allocated to that service, rather than the host server's absolute capacity. Furthermore, we implemented the necessary primitives to support dynamic resource reallocation and server frequency scaling during runtime. These extensions expose control hooks to external scheduling algorithms, allowing them to dynamically manage resource quotas and processing frequencies, thereby directly and accurately impacting the calculation of task processing times in the simulation.

7.5 Energy Management and Stateful Monitoring

The baseline YAFS framework lacks native support for power and energy consumption profiling. To accurately simulate resource-constrained MEC environments—particularly those dependent on energy-harvesting infrastructure—stateful telemetry coupled with high-fidelity energy models is strictly required.

Building upon the analytical energy and battery models formulated in preceding chapters, we engineered a dedicated energy management module for the simulator. Rather than operating as an independent discrete-event process, this module is implemented as a synchronous routine invoked at every simulation time step. At each discrete tick, the module evaluates the global system state: it records task execution events, calculates dynamic energy consumption and battery recharge amounts, and updates the residual capacity of all energy-harvesting nodes. Furthermore, it enforces real-time energy constraints: if battery depletion is detected at any node during a time step update, the module commands the simulation engine to forcibly suspend all services deployed on the affected hardware. Telemetry regarding node-level energy consumption and battery fluctuations is logged deterministically at each tick throughout the simulation.

7.6 Summary of Simulator Contributions

This chapter presented the YAFS-MEC framework, an extended simulation environment necessary to validate the resource orchestration algorithms proposed in this thesis. The baseline YAFS engine was fundamentally modified to support the physical realities of energy-harvesting edge infrastructure. The specific technical contributions integrated into the simulator include the following:

Hierarchical Topology Abstraction: The engineering of domain-specific classes (MECs, edge servers, UE) that translate high-level physical infrastructure models into the low-level graph structures required by the simulation core.

Dependency Synchronisation Barrier: The replacement of the baseline asynchronous trigger with a strict conjunctive (AND-logic) local buffer, constraining services within a DAG to remain blocked until all topological predecessors deliver their payloads.

Resource Contention Mechanics: The elimination of the default unbounded compute assumption by enforcing strict compute resource pre-allocation. This enables accurate task duration calculations for concurrent processes and supports dynamic frequency scaling.

Stateful Energy Telemetry: The implementation of a synchronous, tick-based energy management routine that tracks dynamic power consumption, calculates battery recharge amounts from harvesting sources, and enforces real-time hardware suspension upon energy depletion.

Chapter 8

Conclusion

8.1 Summary of Contributions

MEC has emerged as a fundamental pillar of 5G networks, bringing computational resources closer to end-users to support latency-critical and data-intensive applications. Modern applications are increasingly structured as interconnected services, modelled as dependency graphs, which introduces complex execution dependencies. Simultaneously, the imperative for sustainable computing has driven the integration of energy harvesting into MEC systems to reduce carbon footprints and enable off-grid deployments.

This dissertation identified critical challenges at the intersection of complex application scheduling and energy-constrained edge infrastructure. Existing service placement strategies often treat applications as atomic units and predominantly focus on preserving UE battery, neglecting the severe server-side energy footprint and inter-service communication overheads inherent in DAG-based workloads. Furthermore, in dynamic energy-harvesting MEC environments, fluctuating renewable energy supplies and stochastic user demands create frequent supply-demand mismatches. Blindly offloading individual services to the cloud or aggressively scaling down processor frequencies can trigger cas-

cading delays across the entire dependency graph, risking severe violations of SLAs or catastrophic battery depletion.

To address these gaps, this dissertation proposed a suite of dependency-aware resource allocation and scheduling strategies:

- **Dependency-Aware Heuristic Algorithms:** We developed two novel heuristic algorithms, EDEM and DEM, to co-optimize end-to-end latency and server-side dynamic energy consumption for complex applications in static MEC environments.
- **Dynamic Scaling and Offloading:** We introduced the DSO strategy for hybrid-powered MEC systems, which adaptively manages selective cloud offloading and DVFS under strict sub-deadline constraints.
- **Depletion-Safe Online Energy Management:** We presented the DSM scheme for purely renewable, off-grid MEC systems. This robust strategy jointly coordinates service migration and server scaling to mitigate battery outage risks while gracefully handling input prediction noise through data-smoothing filters.
- **Enhanced Simulation Framework:** We significantly extended the YAFS simulation framework to support DAG-based service execution and dynamic energy harvesting models, providing a highly flexible testbed for edge resource orchestration.

8.2 Limitations

While the proposed strategies demonstrate significant improvements in energy-latency co-optimisation, certain methodological limitations remain:

- **Static Environmental Assumptions:** The foundational algorithms evaluated in Chapter 4 operate under the assumption of static user environments. They do not account for high-velocity

user mobility or fluctuating network topologies during execution.

- **Reliance on Critical Path Approximations:** The EDEM algorithm relies on critical path approximation to manage DAG execution, which may oversimplify dynamics in highly parallelised applications or non-uniform workloads. This potentially leads to suboptimal resource allocation on non-critical branches.
- **Vulnerability to Traffic Bursts and Energy Droughts:** While the dynamic algorithms in Chapters 5 and 6 incorporate input filtering to tolerate prediction noise, they still rely on external forecasting models for energy generation and task arrivals. Consequently, the heuristic deadline decomposition can be vulnerable to bursty traffic that causes queuing delays. Furthermore, purely off-grid horizontal migration approaches remain vulnerable to prolonged, network-wide energy droughts where neighbouring nodes cannot offer surplus energy.
- **Algorithmic Benchmarking:** The proposed heuristic approaches and predictive mechanisms have yet to be benchmarked against contemporary, lightweight learning-based scheduling paradigms or exact solvers to precisely quantify the optimality gap.

8.3 Future Work

This dissertation has shown the critical importance of dependency-aware resource orchestration, but dynamic resource scheduling requires further refinement to address the extreme heterogeneity of next-generation networks. Future research should explore the following directions:

- **Mobility-Aware Scheduling:** Integrating high-velocity user mobility into the scheduling framework presents a vital future direction. Developing predictive handover and proactive service migration strategies would support continuous QoS for mobile users moving across different MEC coverage zones without violating strict deadlines.

- **Large-Scale LLM Training and Inference at the Edge:** Deploying and training large-scale foundation models and LLMs directly at the network edge is an increasingly important topic. Adapting the proposed dependency-aware algorithms to handle the massive, distributed computational requirements of LLM training and inference could resolve significant privacy and latency bottlenecks. This will require novel parallelisation strategies that respect the strict energy constraints of energy-harvesting MDCs.
- **Advanced Predictive Modelling:** Enhancing the system with advanced, integrated predictive modelling offers substantial potential. Combining sophisticated weather-based energy forecasting with deep learning models that anticipate stochastic, bursty user traffic patterns would allow the scheduler to proactively reserve resources or adjust battery management strategies well before congestion or power depletion occurs.
- **Integration of Learning-Based Frameworks:** Expanding the scheduling paradigms to incorporate reinforcement learning or stochastic optimisation frameworks could unlock further efficiency gains. Hybrid approaches could combine the rapid execution of the current heuristics with the adaptability of reinforcement learning agents that continuously learn from long-term environmental and traffic trends.

References

- [1] Victor A Adewopo and Nelly Elsayed. “Smart city transportation: Deep learning ensemble approach for traffic accident detection”. In: *IEEE Access* 12 (2024), pp. 59134–59147.
- [2] Ehsan Ahvar, Shohreh Ahvar, Zoltán Ádám Mann, Noel Crespi, Roch Glitho, and Joaquin Garcia-Alfaro. “DECA: A dynamic energy cost and carbon emission-efficient application placement method for edge clouds”. In: *IEEE Access* 9 (2021), pp. 70192–70213.
- [3] Maria Bermudez Arboleda and Atif Shamim. “Self-powered and cost-effective wireless sensor node for air quality monitoring with an optically transparent smart antenna system”. In: *IEEE Sensors Journal* 24.18 (2024), pp. 28457–28471.
- [4] Patricia Arroba, Rajkumar Buyya, Román Cárdenas, José L Risco-Martín, and José M Moya. “Sustainable edge computing: Challenges and future directions”. In: *Software: Practice and Experience* 54.11 (2024), pp. 2272–2296.
- [5] Mohammad Sadegh Aslanpour, Adel N Toosi, Muhammad Aamir Cheema, and Mohan Baruwal Chhetri. “FaasHouse: sustainable serverless edge computing through energy-aware resource scheduling”. In: *IEEE Transactions on Services Computing* 17.4 (2024), pp. 1533–1547.

- [6] Hani Attar, Mohammed Alghanim, Jafar Ababneh, Khosro Rezaee, Ayat Alrosan, and Mohanad A Deif. “B5G applications and emerging services in smart IoT environments”. In: *International Journal of Crowd Science* 9.2 (2025), pp. 79–95.
- [7] Sadoon Azizi, Mohammad Shojafar, Pedram Farzin, and Javad Dogani. “DCSP: A delay and cost-aware service placement and load distribution algorithm for IoT-based fog networks”. In: *Computer communications* 215 (2024), pp. 9–20.
- [8] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *science* 286.5439 (1999), pp. 509–512.
- [9] Marlom Bey, Pratyay Kuila, Banavath Balaji Naik, and Santanu Ghosh. “Quantum-inspired particle swarm optimization for efficient IoT service placement in edge computing systems”. In: *Expert Systems with Applications* 236 (2024), p. 121270.
- [10] Tathagata Bhattacharya, Mostafa Rahgouy, Xiaopu Peng, Taha Takreeti, Ting Cao, Jianzhou Mao, Amit Das, Xiao Qin, and Apurba Sinha. “Capping carbon emission from green data centers”. In: *International Journal of Energy and Environmental Engineering* 14.4 (2023), pp. 627–641.
- [11] Jing Bi, Ziqi Wang, Haitao Yuan, Jia Zhang, and Mengchu Zhou. “Cost-minimized computation offloading and user association in hybrid cloud and edge computing”. In: *IEEE Internet of Things Journal* 11.9 (2024), pp. 16672–16683.
- [12] Jing Bi, Haitao Yuan, Shuaifei Duanmu, MengChu Zhou, and Abdullah Abusorrah. “Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization”. In: *IEEE Internet of Things Journal* 8.5 (2020), pp. 3774–3785.

- [13] Xincheng Cai, Hongyu Kuang, Hao Hu, Wei Song, and Jian Lü. “Response time aware operator placement for complex event processing in edge computing”. In: *International Conference on Service-Oriented Computing*. Springer. 2018, pp. 264–278.
- [14] Alessandro Cammarano, Chiara Petrioli, and Dora Spenza. “Pro-Energy: A novel energy prediction model for solar and wind energy-harvesting wireless sensor networks”. In: *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. IEEE. 2012, pp. 75–83.
- [15] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. “Run-time adaptation of data stream processing systems: The state of the art”. In: *ACM Computing Surveys* 54.11s (2022), pp. 1–36.
- [16] Carlo Centofanti, Walter Tiberti, Andrea Marotta, Fabio Graziosi, and Dajana Cassioli. “Taming latency at the edge: A user-aware service placement approach”. In: *Computer Networks* 247 (2024), p. 110444.
- [17] Nisha Chaurasia, Mohit Kumar, Rashmi Chaudhry, and Om Prakash Verma. “Comprehensive survey on energy-aware server consolidation techniques in cloud computing”. In: *The Journal of Supercomputing* 77.10 (2021), pp. 11682–11737.
- [18] Huangke Chen, Xiaomin Zhu, Guipeng Liu, and Witold Pedrycz. “Uncertainty-aware online scheduling for real-time workflows in cloud service environment”. In: *IEEE Transactions on Services Computing* 14.4 (2018), pp. 1167–1178.
- [19] Jiawen Chen, Yajun Yang, Chenyang Wang, Heng Zhang, Chao Qiu, and Xiaofei Wang. “Multitask offloading strategy optimization based on directed acyclic graphs for edge computing”. In: *IEEE Internet of Things Journal* 9.12 (2021), pp. 9367–9378.

- [20] Jiechen Chen, Hong Xing, Xiaohui Lin, Arumugam Nallanathan, and Suzhi Bi. “Joint resource allocation and cache placement for location-aware multi-user mobile-edge computing”. In: *IEEE Internet of Things Journal* 9.24 (2022), pp. 25698–25714.
- [21] Jinchao Chen, Pengcheng Han, Ying Zhang, Tao You, and Pengyi Zheng. “Scheduling energy consumption-constrained workflows in heterogeneous multi-processor embedded systems”. In: *Journal of Systems Architecture* 142 (2023), p. 102938.
- [22] Jing Chen, Madhavan Manivannan, Bhavishya Goel, Mustafa Abduljabbar, and Miquel Pericàs. “Steer: Asymmetry-aware energy efficient task scheduler for cluster-based multicore architectures”. In: *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2022, pp. 326–335.
- [23] Long Chen, Jigang Wu, Jun Zhang, Hong-Ning Dai, Xin Long, and Mianyang Yao. “Dependency-aware computation offloading for mobile edge computing with edge-cloud cooperation”. In: *IEEE Transactions on Cloud Computing* 10.4 (2020), pp. 2451–2468.
- [24] Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “Dynamic Service Scheduling and Resource Management in Energy-Harvesting Multi-Access Edge Computing”. In: *2025 IEEE International Conferences on Internet of Things (iThings) IEEE Green Computing & Communications (GreenCom) IEEE Cyber, Physical & Social Computing (CPSCoM) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE. 2025, pp. 122–130.
- [25] Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “Efficient Placement of Interdependent Services in Multi-access Edge Computing”. In: *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer. 2024, pp. 150–164.

- [26] Shuyi Chen, Panagiotis Oikonomou, Zhengchang Hua, Nikos Tziritas, Karim Djemame, Nan Zhang, and Georgios Theodoropoulos. “QoS-aware placement of interdependent services in energy-harvesting-enabled multi-access edge computing”. In: *Future Generation Computer Systems* (2025), p. 108009.
- [27] Xiaojing Chen, Si Chen, Wei Ni, Xin Wang, Sihai Zhang, Shunqing Zhang, Yanzan Sun, Shugong Xu, and Abbas Jamalipour. “Optimal two-timescale configuration of mobile edge computing with mixed energy supply”. In: *IEEE Transactions on Smart Grid* 15.5 (2024), pp. 4765–4778.
- [28] Xiaojing Chen, Hanfei Wen, Wei Ni, Shunqing Zhang, Xin Wang, Shugong Xu, and Qingqi Pei. “Distributed online optimization of edge computing with mixed power supply of renewable energy and smart grid”. In: *IEEE Transactions on Communications* 70.1 (2021), pp. 389–403.
- [29] Carla-Fabiana Chiasserini and Ramesh R Rao. “Improving energy saving in wireless systems by using dynamic power management”. In: *IEEE Transactions on wireless communications* 2.5 (2003), pp. 1090–1100.
- [30] Chih-Hsun Chou, Daniel Wong, and Laxmi N Bhuyan. “Dynsleep: Fine-grained power management for a latency-critical data center application”. In: *Proceedings of the 2016 international symposium on low power electronics and design*. 2016, pp. 212–217.
- [31] Wu Chia Chuan, Shams Ul Arfeen Laghari, Selvakumar Manickam, Erum Ashraf, and Shankar Karuppayah. “Challenges and opportunities in fog computing scheduling: a literature review”. In: *IEEE Access* 13 (2025), pp. 14702–14726.

- [32] Arthur Clerjon and Fabien Perdu. “Matching intermittent electricity supply and demand with electricity storage-An optimization based on a time scale analysis”. In: *Energy* 241 (2022), p. 122799.
- [33] Pedro Cruz, Nadjib Achir, and Aline Carneiro Viana. “On the edge of the deployment: A survey on multi-access edge computing”. In: *ACM Computing Surveys* 55.5 (2022), pp. 1–34.
- [34] Howard David, Chris Fallin, Eugene Gorbatoov, Ulf R Hanebutte, and Onur Mutlu. “Memory power management via dynamic voltage/frequency scaling”. In: *Proceedings of the 8th ACM international conference on Autonomic computing*. 2011, pp. 31–40.
- [35] Thinh Quang Dinh, Quang Duy La, Tony QS Quek, and Hyundong Shin. “Learning for computation offloading in mobile edge computing”. In: *IEEE Transactions on Communications* 66.12 (2018), pp. 6353–6367.
- [36] Thembelihle Dlamini and Sifiso Vilakati. “Remote and rural connectivity: Infrastructure and resource sharing principles”. In: *Wireless Communications and Mobile Computing* 2021.1 (2021), p. 6065119.
- [37] Kazim Ergun, Raid Ayoub, Pietro Mercati, and Tajana Šimunić Rosing. “Dynamic reliability management of multigateway IoT edge computing systems”. In: *IEEE Internet of Things Journal* 10.5 (2022), pp. 3864–3889.
- [38] Melike Erol-Kantarci and Sukhmani Sukhmani. “Caching and computing at the edge for mobile augmented reality and virtual reality (AR/VR) in 5G”. In: *Ad Hoc Networks: 9th International Conference, AdHocNets 2017, Niagara Falls, ON, Canada, September 28–29, 2017, Proceedings*. Springer. 2018, pp. 169–177.

- [39] MECISG ETSI. “Multi-access edge computing (mec) framework and reference architecture”. In: *ETSI GS MEC 3.2* (2019).
- [40] Mohammad Faraji-Mehmandar, Mostafa Ghobaei-Arani, and Ali Shakarami. “A cost-aware IoT application deployment approach in fog computing”. In: *Cluster Computing* 28.3 (2025), p. 199.
- [41] Vajiheh Farhadi, Fidan Mehmeti, Ting He, Thomas F La Porta, Hana Khamfroush, Shiqiang Wang, Kevin S Chan, and Konstantinos Poularakis. “Service placement and request scheduling for data-intensive applications in edge clouds”. In: *IEEE/ACM Transactions on Networking* 29.2 (2021), pp. 779–792.
- [42] Paris Flegkas, Giannis Roumpos, Vasilis Sourlas, and Angelos Amditis. “Edge to Cloud Service Placement based on Reinforcement Learning in 6G Networks”. In: *ICC 2025-IEEE International Conference on Communications*. IEEE. 2025, pp. 1139–1144.
- [43] Zhifan Gao, Heye Zhang, Shizhou Dong, Shanhui Sun, Xin Wang, Guang Yang, Wanqing Wu, Shuo Li, and Victor Hugo C de Albuquerque. “Salient object detection in the distributed cloud-edge intelligent network”. In: *IEEE Network* 34.2 (2020), pp. 216–224.
- [44] Kaouther Gasmi, Selma Dilek, Suleyman Tosun, and Suat Ozdemir. “A survey on computation offloading and service placement in fog computing-based IoT”. In: *the Journal of Supercomputing* 78.2 (2022), pp. 1983–2014.
- [45] Yazeed Yasin Ghadi, Syed Faisal Abbas Shah, Tehseen Mazhar, Tariq Shahzad, Khmaies Ouahada, and Habib Hamam. “Enhancing patient healthcare with mobile edge computing and 5G: challenges and solutions for secure online health tools”. In: *Journal of Cloud Computing* 13.1 (2024), p. 93.

- [46] Ardalan Ghasemzadeh, Hadi S Aghdasi, and Saeed Saeedvand. “Edge server placement and allocation optimization: a tradeoff for enhanced performance”. In: *Cluster Computing* 27.5 (2024), pp. 5783–5797.
- [47] Lin Gu, Weiying Zhang, Zhongkui Wang, Deze Zeng, and Hai Jin. “Service management and energy scheduling toward low-carbon edge computing”. In: *IEEE Transactions on Sustainable Computing* 8.1 (2022), pp. 109–119.
- [48] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296.
- [49] Davit Harutyunyan, Nashid Shahriar, Raouf Boutaba, and Roberto Riggio. “Latency and mobility-aware service function chain placement in 5G networks”. In: *IEEE Transactions on Mobile Computing* 21.5 (2020), pp. 1697–1709.
- [50] Hiwa Omer Hassan, Sadoon Azizi, and Mohammad Shojafar. “Priority, network and energy-aware placement of IoT-based application services in fog-cloud environments”. In: *IET communications* 14.13 (2020), pp. 2117–2129.
- [51] Najmul Hassan, Kok-Lim Alvin Yau, and Celimuge Wu. “Edge computing in 5G: A review”. In: *IEEE access* 7 (2019), pp. 127276–127289.
- [52] Han Hu, Qun Wang, Rose Qingyang Hu, and Hongbo Zhu. “Mobility-aware offloading and resource allocation in a MEC-enabled IoT network with energy harvesting”. In: *IEEE Internet of Things Journal* 8.24 (2021), pp. 17541–17556.

- [53] Yi Hu, Xi Liao, Junhui Hu, Hao Wen, Huan Xu, Kai Peng, and Menglan Hu. “Energy-Efficient Microservice Orchestration for Latency-Sensitive Applications in Edge Computing: A Reinforcement Learning Approach”. In: *IEEE Internet of Things Journal* 13.2 (2025), pp. 2035–2052.
- [54] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. “Mobile edge computing—A key technology towards 5G”. In: *ETSI white paper* 11.11 (2015), pp. 1–16.
- [55] Tiansheng Huang, Weiwei Lin, Chennian Xiong, Rui Pan, and Jingxuan Huang. “An ant colony optimization-based multiobjective service replicas placement strategy for fog computing”. In: *IEEE Transactions on Cybernetics* 51.11 (2020), pp. 5595–5608.
- [56] Nathaniel Hudson, Hana Khamfroush, Matt Baughman, Daniel E Lucani, Kyle Chard, and Ian Foster. “QoS-aware edge AI placement and scheduling with multiple implementations in FaaS-based edge computing”. In: *Future Generation Computer Systems* 157 (2024), pp. 250–263.
- [57] Samia Ijaz, Ehsan Ullah Munir, Saima Gulzar Ahmad, M Mustafa Rafique, and Omer F Rana. “Energy-makespan optimization of workflow scheduling in fog–cloud computing”. In: *Computing* 103 (2021), pp. 2033–2059.
- [58] Hengli Jin, Mark A Gregory, and Shuo Li. “A review of intelligent computation offloading in multiaccess edge computing”. In: *IEEE Access* 10 (2022), pp. 71481–71495.
- [59] Ferdos Kazemi, Behnam Barzegar, Homayun Motameni, and Meisam Yadollahzadeh-Tabari. “An energy-aware scheduling in DVFS-enabled heterogeneous edge computing environments”. In: *The Journal of Supercomputing* 81.9 (2025), p. 1078.

- [60] Muhammad Asif Khan, Emna Baccour, Zina Chkirbene, Aiman Erbad, Ridha Hamila, Mounir Hamdi, and Moncef Gabbouj. “A survey on mobile edge computing for video streaming: Opportunities and challenges”. In: *IEEE Access* 10 (2022), pp. 120514–120550.
- [61] Shweta Khare, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Aniruddha Gokhale, Yogesh Barve, Anirban Bhattacharjee, and Xenofon Koutsoukos. “Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs”. In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 2019, pp. 1–14.
- [62] Arzad Alam Kherani et al. “Development of mec system for indigenous 5g test-bed”. In: *2021 International Conference on COMMunication Systems & NETWORKS (COMSNETS)*. IEEE. 2021, pp. 131–133.
- [63] Soomin Ki, Gyuri Byun, Kyungwoon Cho, and Hyokyung Bahn. “Co-optimizing CPU voltage, memory placement, and task offloading for energy-efficient mobile systems”. In: *IEEE Internet of Things Journal* 10.10 (2023), pp. 9177–9192.
- [64] Yu-Jen Ku, Po-Han Chiang, and Sujit Dey. “Real-time QoS optimization for vehicular edge computing with off-grid roadside units”. In: *IEEE Transactions on Vehicular Technology* 69.10 (2020), pp. 11975–11991.
- [65] Isaac Lera, Carlos Guerrero, and Carlos Juiz. “YAFS: A simulator for IoT scenarios in fog computing”. In: *IEEE Access* 7 (2019), pp. 91745–91758.
- [66] Rui Li, Zhi Zhou, Xiaoxi Zhang, and Xu Chen. “Joint application placement and request routing optimization for dynamic edge computing service management”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4581–4596.

- [67] Wei Li, Ting Yang, Flavia C Delicato, Paulo F Pires, Zahir Tari, Samee U Khan, and Albert Y Zomaya. “On enabling sustainable edge computing with renewable energy resources”. In: *IEEE communications magazine* 56.5 (2018), pp. 94–101.
- [68] Yuqing Li, Wenkuan Dai, Xiaoying Gan, Haiming Jin, Luoyi Fu, Huadong Ma, and Xinbing Wang. “Cooperative service placement and scheduling in edge clouds: A deadline-driven approach”. In: *IEEE Transactions on Mobile Computing* 21.10 (2021), pp. 3519–3535.
- [69] Zezu Liang, Yuan Liu, Tat-Ming Lok, and Kaibin Huang. “Multi-cell mobile edge computing: Joint service migration and resource allocation”. In: *IEEE Transactions on Wireless Communications* 20.9 (2021), pp. 5898–5912.
- [70] Miao Lin, Jianqing Xi, Weihua Bai, and Jiayin Wu. “Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud”. In: *IEEE access* 7 (2019), pp. 83088–83100.
- [71] Zhen Lin, Liming Lu, Jianping Shuai, Hong Zhao, and Ali Shahidinejad. “An efficient and autonomous planning scheme for deploying IoT services in fog computing: A metaheuristic-based approach”. In: *IEEE Transactions on Computational Social Systems* 11.1 (2023), pp. 1415–1429.
- [72] Qifeng Liu, Qilin Fan, Xu Zhang, Xiuhua Li, Kai Wang, and Qingyu Xiong. “DRL-based time-varying workload scheduling with priority and resource awareness”. In: *IEEE Transactions on Network and Service Management* (2025).
- [73] Tong Liu, Shenggang Ni, Xiaoqiang Li, Yanmin Zhu, Linghe Kong, and Yuanyuan Yang. “Deep reinforcement learning based approach for online service placement and computation resource allocation in edge computing”. In: *IEEE Transactions on Mobile Computing* 22.7 (2022), pp. 3870–3881.

- [74] Yicen Liu and Junning Zhang. “Service function chain embedding meets machine learning: Deep reinforcement learning approach”. In: *IEEE Transactions on Network and Service Management* 21.3 (2024), pp. 3465–3481.
- [75] V Srinivas Lokavarapu, Kunjam Nageswara Rao, Sitaratnam Gokuruboyina, and Shiva Shankar Reddy. “An in-depth analysis of computation offloading approaches in multi-access edge computing systems”. In: *Peer-to-Peer Networking and Applications* 18.4 (2025), p. 219.
- [76] Man Lu, Guifang Fu, Nisreen Beshir Osman, and Usama Konbr. “Green energy harvesting strategies on edge-based urban computing in sustainable internet of things”. In: *Sustainable cities and society* 75 (2021), p. 103349.
- [77] Wei Lu, Xianyu Meng, and Guanfei Guo. “Fast service migration method based on virtual machine technology for MEC”. In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4344–4354.
- [78] Qu Yuan Luo, Shihong Hu, Changle Li, Guanghui Li, and Weisong Shi. “Resource scheduling in edge computing: A survey”. In: *IEEE communications surveys & tutorials* 23.4 (2021), pp. 2131–2165.
- [79] Mingsong Lv and Enyu Xu. “Deep learning on energy harvesting IoT devices: Survey and future challenges”. In: *IEEE Access* 10 (2022), pp. 124999–125014.
- [80] Wenkai Lv, Pengfei Yang, Tianyang Zheng, Bijie Yi, Yunqing Ding, Quan Wang, and Minwen Deng. “Energy consumption and qos-aware co-offloading for vehicular edge computing”. In: *IEEE Internet of Things Journal* 10.6 (2022), pp. 5214–5225.

- [81] Huirong Ma, Peng Huang, Zhi Zhou, Xiaoxi Zhang, and Xu Chen. “GreenEdge: Joint green energy scheduling and dynamic task offloading in multi-tier edge computing systems”. In: *IEEE Transactions on Vehicular Technology* 71.4 (2022), pp. 4322–4335.
- [82] Huirong Ma, Zhi Zhou, and Xu Chen. “Predictive service placement in mobile edge computing”. In: *2019 IEEE/CIC International Conference on Communications in China (ICCC)*. IEEE. 2019, pp. 792–797.
- [83] Xiao Ma, Ao Zhou, Shan Zhang, Qing Li, Alex X Liu, and Shangguang Wang. “Dynamic task scheduling in cloud-assisted mobile edge computing”. In: *IEEE Transactions on Mobile Computing* 22.4 (2021), pp. 2116–2130.
- [84] Xiaojin Ma, Huahu Xu, Honghao Gao, and Minjie Bian. “Real-time multiple-workflow scheduling in cloud environments”. In: *IEEE Transactions on Network and Service Management* 18.4 (2021), pp. 4002–4018.
- [85] Mobasshir Mahbub and Raed M Shubair. “Contemporary advances in multi-access edge computing: A survey of fundamentals, architecture, technologies, deployment cases, security, challenges, and directions”. In: *Journal of Network and Computer Applications* 219 (2023), p. 103726.
- [86] Ayeh Mahjoubi, Javid Taheri, Karl-Johan Grinnemo, and Shuiguang Deng. “Optimal placement of recurrent service chains on distributed edge-cloud infrastructures”. In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE. 2021, pp. 495–502.
- [87] Hadi Tabatabaee Malazi, Saqib Rasool Chaudhry, Aqeel Kazmi, Andrei Palade, Christian Cabrera, Gary White, and Siobhán Clarke. “Dynamic service placement in multi-access edge computing: A systematic literature review”. In: *IEEE Access* 10 (2022), pp. 32639–32688.

- [88] Ramachandran Manikandan, Rizwan Patan, Amir H Gandomi, Perumal Sivanesan, and Har-
iharan Kalyanaraman. “Hash polynomial two factor decision tree using IoT for smart health
care scheduling”. In: *Expert Systems with Applications* 141 (2020), p. 112924.
- [89] Mahshid Mehrabi, Shiwei Shen, Vincent Latzko, Yuanfei Wang, and Frank HP Fitzek. “Energy-
aware cooperative offloading framework for inter-dependent and delay-sensitive tasks”. In:
GLOBECOM 2020-2020 IEEE Global Communications Conference. IEEE. 2020, pp. 1–6.
- [90] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3
(2017), e103.
- [91] Jiwei Mo, Jiangshu Liu, and Zhiwei Zhao. “Exploiting function-level dependencies for task
offloading in edge computing”. In: *IEEE INFOCOM 2022-IEEE Conference on Computer
Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2022, pp. 1–6.
- [92] Rajeev Muralidhar, Renata Borovica-Gajic, and Rajkumar Buyya. “Energy efficient comput-
ing systems: Architectures, abstractions and modeling to techniques and standards”. In: *ACM
Computing Surveys (CSUR)* 54.11s (2022), pp. 1–37.
- [93] Ehzaz Mustafa, Junaid Shuja, S Khaliq uz Zaman, Ali Imran Jehangiri, Sadia Din, Faisal
Rehman, Saad Mustafa, Tahir Maqsood, and Abdul Nasir Khan. “Joint wireless power transfer
and task offloading in mobile edge computing: a survey”. In: *Cluster Computing* 25.4 (2022),
pp. 2429–2448.
- [94] Pranjal Kumar Nandi, Md Rejaul Islam Reaj, Sujan Sarker, Md Abdur Razzaque, Md Mamun-
or Rashid, and Palash Roy. “Task offloading to edge cloud balancing utility and cost for energy
harvesting internet of things”. In: *Journal of network and computer applications* 221 (2024),
p. 103766.

- [95] Giovanni Nardini, Dario Sabella, Giovanni Stea, Purvi Thakkar, and Antonio Virdis. “Simu5g—an omnet++ library for end-to-end performance evaluation of 5g networks”. In: *IEEE Access* 8 (2020), pp. 181176–181191.
- [96] Gianfranco Nencioni, Rosario G Garroppo, and Ruxandra F Olimid. “5G multi-access edge computing: A survey on security, dependability, and performance”. In: *IEEE Access* 11 (2023), pp. 63496–63533.
- [97] Zeinab Nezami, Kamran Zamanifar, Karim Djemame, and Evangelos Pournaras. “Decentralized edge-to-cloud load balancing: Service placement for the Internet of Things”. In: *Ieee Access* 9 (2021), pp. 64983–65000.
- [98] Loc X Nguyen, Yan Kyaw Tun, Tri Nguyen Dang, Yu Min Park, Zhu Han, and Choong Seon Hong. “Dependency tasks offloading and communication resource allocation in collaborative UAVs networks: A meta-heuristic approach”. In: *IEEE Internet of Things Journal* (2023).
- [99] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Shupeng Wang, Xiping Hu, Song Guo, Tie Qiu, Bin Hu, and Ricky YK Kwok. “Distributed and dynamic service placement in pervasive edge computing networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.6 (2020), pp. 1277–1292.
- [100] Panagiotis Oikonomou, Maria G Koziri, Nikos Tziritas, Antonios N Dadaliaris, Thanasis Loukopoulos, Georgios I Stamoulis, and Samee U Khan. “Scheduling video transcoding jobs in the cloud”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 442–449.

- [101] Tao Ouyang, Zhi Zhou, and Xu Chen. “Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing”. In: *IEEE Journal on Selected Areas in Communications* 36.10 (2018), pp. 2333–2345.
- [102] Samodha Pallewatta, Vassilis Kostakos, and Rajkumar Buyya. “Reliability-aware proactive placement of microservices-based IoT applications in fog computing environments”. In: *IEEE Transactions on Mobile Computing* 23.12 (2024), pp. 11326–11341.
- [103] Saroj Kumar Panda, Man Lin, and Ti Zhou. “Energy-efficient computation offloading with DVFS using deep reinforcement learning for time-critical IoT applications in edge computing”. In: *IEEE Internet of Things Journal* 10.8 (2022), pp. 6611–6621.
- [104] Suraj Singh Panwar, Man Mohan Singh Rauthan, and Varun Barthwal. “A systematic review on effective energy utilization management strategies in cloud data centers”. In: *Journal of Cloud Computing* 11.1 (2022), p. 95.
- [105] Giovanni Perin, Michele Berno, Tomaso Erseghe, and Michele Rossi. “Towards sustainable edge computing through renewable energy resources and online, distributed and predictive scheduling”. In: *IEEE Transactions on Network and Service Management* 19.1 (2021), pp. 306–321.
- [106] Padmanabhan Pillai and Kang G Shin. “Real-time dynamic voltage scaling for low-power embedded operating systems”. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001, pp. 89–102.
- [107] Gaetano Francesco Pittalà, Lorenzo Rinieri, Amir Al Sadi, Gianluca Davoli, Andrea Melis, Marco Prandini, and Walter Cerroni. “Leveraging data plane programmability to enhance service orchestration at the edge: A focus on industrial security”. In: *Computer Networks* 246 (2024), p. 110397.

- [108] Gopika Premasankar and Bissan Ghaddar. “Energy-efficient service placement for latency-sensitive applications in edge computing”. In: *IEEE internet of things journal* 9.18 (2022), pp. 17926–17937.
- [109] Yazdan Ahmad Qadri, Haejoon Jung, and Dusit Niyato. “Toward the internet of medical things for real-time health monitoring over Wi-Fi”. In: *IEEE Network* 38.5 (2024), pp. 229–237.
- [110] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. “Mobile edge intelligence for large language models: A contemporary survey”. In: *IEEE Communications Surveys & Tutorials* 27.6 (2025), pp. 3820–3860.
- [111] Samuel Rac and Mats Brorsson. “Cost-aware service placement and scheduling in the edge-cloud continuum”. In: *ACM Transactions on Architecture and Code Optimization* 21.2 (2024), pp. 1–24.
- [112] Mostafa Raeisi-Varzaneh, Omar Dakkak, Adib Habbal, and Byung-Seo Kim. “Resource scheduling in edge computing: Architecture, taxonomy, open issues and future research directions”. In: *IEEE access* 11 (2023), pp. 25329–25350.
- [113] Rajkumar Rajavel, Sathish Kumar Ravichandran, Karthikeyan Harimoorthy, Partheeban Nagappan, and Kanagachidambaresan Ramasubramanian Gobichettipalayam. “IoT-based smart healthcare video surveillance system using edge computing”. In: *Journal of ambient intelligence and humanized computing* 13.6 (2022), pp. 3195–3207.
- [114] Daler Rakhmatov and Sarma Vrudhula. “Energy management for battery-powered embedded systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 2.3 (2003), pp. 277–324.

- [115] Pasika Ranaweera, Chamitha De Alwis, Anca D Jurcut, and Madhusanka Liyanage. “Realizing contact-less applications with multi-access edge computing”. In: *ICT Express* 8.4 (2022), pp. 575–587.
- [116] Nayereh Rasouli, Cristian Klein, and Erik Elmroth. “Resource management for mission-critical applications in edge computing: systematic review on recent research and open issues”. In: *ACM Computing Surveys* 58.3 (2025), pp. 1–37.
- [117] George F Riley and Thomas R Henderson. “The ns-3 network simulator”. In: *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.
- [118] Soheil Rostami, Hoang Duy Trinh, Sandra Lagen, Mario Costa, Mikko Valkama, and Paolo Dini. “Wake-up scheduling for energy-efficient mobile devices”. In: *IEEE Transactions on Wireless Communications* 19.9 (2020), pp. 6020–6036.
- [119] Kuheli Sai and David Tipper. “Sustainability and Power Outage-aware Placement of Edge Computing in NextG RANs”. In: *2024 IEEE International Symposium on Technology and Society (ISTAS)*. IEEE. 2024, pp. 1–8.
- [120] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. “An overview of service placement problem in fog and edge computing”. In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–35.
- [121] Indranil Sarkar, Mainak Adhikari, Neeraj Kumar, and Sanjay Kumar. “Dynamic task placement for deadline-aware IoT applications in federated fog networks”. In: *IEEE Internet of Things Journal* 9.2 (2021), pp. 1469–1478.

- [122] Nazanin Sarrafzade, Reza Entezari-Maleki, and Leonel Sousa. “A genetic-based approach for service placement in fog computing”. In: *The Journal of Supercomputing* 78.8 (2022), pp. 10854–10875.
- [123] Seyed Salar Sefati and Simona Halunga. “Ultra-reliability and low-latency communications on the internet of things based on 5G network: literature review, classification, and future research view”. In: *Transactions on emerging telecommunications technologies* 34.6 (2023), e4770.
- [124] Henda Sfaxi, Imene Lahyani, Sami Yangui, and Mouna Torjmen. “Latency-aware and proactive service placement for edge computing”. In: *IEEE Transactions on Network and Service Management* 21.4 (2024), pp. 4243–4254.
- [125] Faisal Karim Shaikh, Sherali Zeadally, and Ernesto Exposito. “Enabling technologies for green internet of things”. In: *IEEE Systems Journal* 11.2 (2015), pp. 983–994.
- [126] Ying Shang, Jinglei Li, and Xiguang Wu. “Dag-based task scheduling in mobile edge computing”. In: *2020 7th International Conference on Information Science and Control Engineering (ICISCE)*. IEEE. 2020, pp. 426–431.
- [127] Qiaoqiao Shen, Bin-Jie Hu, and Enjun Xia. “Dependency-aware task offloading and service caching in vehicular edge computing”. In: *IEEE Transactions on Vehicular Technology* 71.12 (2022), pp. 13182–13197.
- [128] Jie Shi, Zuzhao Ye, H Oliver Gao, and Nanpeng Yu. “Lyapunov optimization in online battery energy storage system control for commercial buildings”. In: *IEEE Transactions on Smart Grid* 14.1 (2022), pp. 328–340.

- [129] Yushan Siriwardhana, Pawani Porambage, Madhusanka Liyanage, and Mika Ylianttila. “A survey on mobile augmented reality with 5G mobile edge computing: Architectures, applications, and technical aspects”. In: *IEEE Communications Surveys & Tutorials* 23.2 (2021), pp. 1160–1192.
- [130] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. “Optimized IoT service placement in the fog”. In: *Service Oriented Computing and Applications* 11.4 (2017), pp. 427–443.
- [131] Yonglak Son, Udit Gupta, Andrew McCrabb, Young Geun Kim, Valeria Bertacco, David Brooks, and Carole-Jean Wu. “Greenscale: Carbon optimization for edge computing”. In: *IEEE Internet of Things Journal* (2025).
- [132] Xifei Song, Jie Feng, Lei Liu, Qingqi Pei, F Richard Yu, and Ning Zhang. “A Deep Reinforcement Learning With Transformer Integration for Directed Acyclic Graph Scheduling in Edge Networks”. In: *IEEE Transactions on Wireless Communications* (2025).
- [133] Balázs Sonkoly, János Czentye, Márk Szalay, Balázs Németh, and László Toka. “Survey on placement methods in the edge and beyond”. In: *IEEE Communications Surveys & Tutorials* 23.4 (2021), pp. 2590–2629.
- [134] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. “Edgecloudsim: An environment for performance evaluation of edge computing systems”. In: *Transactions on Emerging Telecommunications Technologies* 29.11 (2018), e3493.
- [135] Meeniga Sriraghavendra, Priyanka Chawla, Huaming Wu, Sukhpal Singh Gill, and Rajkumar Buyya. “DoSP: A deadline-aware dynamic service placement algorithm for workflow-oriented IoT applications in fog-cloud computing environments”. In: *Energy conservation solutions for fog-edge computing paradigms*. Springer, 2021, pp. 21–47.

- [136] Ali Sunyaev. “The internet of things”. In: *Internet computing: Principles of distributed systems and emerging internet-based technologies*. Springer, 2024, pp. 317–352.
- [137] Masoud Taghavian, Yassine Hadjadj-Aoul, Géraldine Texier, Nicolas Huin, and Philippe Bertin. “An approach to network service placement reconciling optimality and scalability”. In: *IEEE Transactions on Network and Service Management* 20.3 (2023), pp. 2218–2229.
- [138] Haruto Taka, Fujun He, and Eiji Oki. “Service placement and user assignment in multi-access edge computing with base-station failure”. In: *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*. IEEE. 2022, pp. 1–10.
- [139] Imane Taleb, Jean-Loup Guillaume, and Benjamin Duthil. “A survey on services placement algorithms in integrated cloud-fog/edge computing”. In: *ACM Computing Surveys* 57.11 (2025), pp. 1–36.
- [140] Anum Talpur and Mohan Gurusamy. “Drld-sp: A deep-reinforcement-learning-based dynamic service placement in edge-enabled internet of vehicles”. In: *IEEE Internet of Things Journal* 9.8 (2021), pp. 6239–6251.
- [141] Chaogang Tang, Yubin Zhao, and Huaming Wu. “Lyapunov-guided optimal service placement in vehicular edge computing”. In: *China Communications* 20.3 (2023), pp. 201–217.
- [142] United States Environmental Protection Agency (EPA). *Emissions & Generation Resource Integrated Database (eGRID)*. <https://www.epa.gov/egrid>. 2023 (accessed 14 April 2025).
- [143] András Varga. “Discrete event simulation system”. In: *Proc. of the European Simulation Multiconference (ESM’2001)*. Vol. 17. 2001.

- [144] Omar Abdel Wahab, Nadjia Kara, Claes Edstrom, and Yves Lemieux. “MAPLE: A machine learning approach for efficient placement and adjustment of virtual network functions”. In: *Journal of Network and Computer Applications* 142 (2019), pp. 37–50.
- [145] Chengzhi Wang, Peng Yang, Jiawei Hou, Zhi Liu, and Ning Zhang. “Dependence-aware multitask scheduling for edge video analytics with accuracy guarantee”. In: *IEEE Internet of Things Journal* 11.16 (2024), pp. 26970–26983.
- [146] Liangyuan Wang, Xudong Liu, Haonan Ding, Yi Hu, Kai Peng, and Menglan Hu. “Energy-delay-aware joint microservice deployment and request routing with DVFS in edge: a reinforcement learning approach”. In: *IEEE Transactions on Computers* 74.5 (2025), pp. 1589–1604.
- [147] Wendong Wang, Cheng Feng, Bo Zhang, and Hui Gao. “Environmental monitoring based on fog computing paradigm and internet of things”. In: *IEEE access* 7 (2019), pp. 127154–127165.
- [148] Yanting Wang, Min Sheng, Xijun Wang, Liang Wang, and Jiandong Li. “Mobile-edge computing: Partial computation offloading using dynamic voltage scaling”. In: *IEEE Transactions on Communications* 64.10 (2016), pp. 4268–4282.
- [149] Duncan J Watts and Steven H Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *nature* 393.6684 (1998), pp. 440–442.
- [150] Qiufen Xia, Guijie Wang, Zichuan Xu, Weifa Liang, and Zhou Xu. “Efficient algorithms for service chaining in NFV-enabled satellite edge networks”. In: *IEEE Transactions on Mobile Computing* 23.5 (2023), pp. 5677–5694.

- [151] Shichao Xia, Zhixiu Yao, Yun Li, and Shiwen Mao. “Online distributed offloading and computing resource management with energy harvesting for heterogeneous MEC-enabled IoT”. In: *IEEE transactions on wireless communications* 20.10 (2021), pp. 6743–6757.
- [152] Chen Xu, Mengzhuo Lv, Kun Zhang, Kui Cao, Gang Wang, Mingzhu Wei, and Bei Peng. “Energy consumption and time-delay optimization of dependency-aware tasks offloading for industry 5.0 applications”. In: *IEEE Transactions on Consumer Electronics* 70.1 (2023), pp. 1590–1600.
- [153] Jie Xu, Lixing Chen, and Shaolei Ren. “Online learning for offloading and autoscaling in energy harvesting mobile edge computing”. In: *IEEE Transactions on Cognitive Communications and Networking* 3.3 (2017), pp. 361–373.
- [154] Minzhi Xu, Weidong Li, Xuejie Zhang, and Qian Su. “A discrete dwarf mongoose optimization algorithm to solve task assignment problems on smart farms”. In: *Cluster Computing* (2024), pp. 1–20.
- [155] Xiaolong Xu, Liang Yao, Muhammad Bilal, Shaohua Wan, Fei Dai, and Kim-Kwang Raymond Choo. “Service migration across edge devices in 6G-enabled Internet of Vehicles networks”. In: *IEEE Internet of Things Journal* 9.3 (2021), pp. 1930–1937.
- [156] Zheyuan Yang, Suzhi Bi, and Ying-Jun Angela Zhang. “Dynamic offloading and trajectory control for UAV-enabled mobile edge computing system with energy harvesting devices”. In: *IEEE Transactions on Wireless Communications* 21.12 (2022), pp. 10515–10528.
- [157] Hao Yu, Tarik Taleb, and Jiawei Zhang. “Deterministic latency/jitter-aware service function chaining over beyond 5G edge fabric”. In: *IEEE Transactions on Network and Service Management* 19.3 (2022), pp. 2148–2162.

- [158] Lingfei Yu, Hongliu Xu, Yunhao Zeng, and Jiali Deng. “Delay-aware resource allocation for partial computation offloading in mobile edge cloud computing”. In: *Pervasive and Mobile Computing* 105 (2024), p. 101996.
- [159] Yi Yue, Bo Cheng, Meng Wang, Biyi Li, Xuan Liu, and Junliang Chen. “Throughput optimization and delay guarantee VNF placement for mapping SFC requests in NFV-enabled networks”. In: *IEEE Transactions on Network and Service Management* 18.4 (2021), pp. 4247–4262.
- [160] Jieyi Zhang, Yongzhi Zhai, Zhang Liu, and Ye Wang. “A Lyapunov Based Resource Allocation Method for Edge-Assisted Industrial Internet of Things”. In: *IEEE Internet of Things Journal* (2024).
- [161] Jun Zhang and Khaled B Letaief. “Mobile edge intelligence and computing for the internet of vehicles”. In: *Proceedings of the IEEE* 108.2 (2019), pp. 246–261.
- [162] Xiaojie Zhang and Saptarshi Debroy. “Resource management in mobile edge computing: A comprehensive survey”. In: *ACM Computing Surveys* 55.13s (2023), pp. 1–37.
- [163] Yanfang Zhang, Jian Chen, Yuchen Zhou, Long Yang, Bingtao He, and Yijin Yang. “Dependent task offloading with energy-latency tradeoff in mobile edge computing”. In: *IET Communications* 16.17 (2022), pp. 1993–2001.
- [164] Ying Zhang, Xin Wu, Yaguo Lei, Junyi Cao, and Wei-Hsin Liao. “Self-powered wireless condition monitoring for rotating machinery”. In: *IEEE Internet of Things Journal* 11.2 (2023), pp. 3095–3107.

-
- [165] Fengjun Zhao, Ying Chen, Yongchao Zhang, Zhiyong Liu, and Xin Chen. “Dynamic offloading and resource scheduling for mobile-edge computing with energy harvesting devices”. In: *IEEE Transactions on Network and Service Management* 18.2 (2021), pp. 2154–2165.
- [166] Jie Zhao, Maria A Rodríguez, and Rajkumar Buyya. “A deep reinforcement learning approach to resource management in hybrid clouds harnessing renewable energy and task scheduling”. In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE. 2021, pp. 240–249.
- [167] Ruijuan Zheng, Junwei Xu, Xueqi Wang, Muhua Liu, and Junlong Zhu. “Service placement strategies in mobile edge computing based on an improved genetic algorithm”. In: *Pervasive and Mobile Computing* 105 (2024), p. 101986.
- [168] Junlong Zhou, Kun Cao, Xiumin Zhou, Mingsong Chen, Tongquan Wei, and Shiyan Hu. “Throughput-conscious energy allocation and reliability-aware task assignment for renewable powered in-situ server systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (2021), pp. 516–529.