Adaptive Resilience of Intelligent Distributed Applications in the Edge-Cloud Environment

A thesis submitted in partial fulfilment

of the requirement for the degree of Doctor of Philosophy

Osama Ahmed S Almurshed

April 2024

Cardiff University
School of Computer Science & Informatics

Abstract

This thesis navigates the complexities of Internet of Things (IoT) application placement in hybrid fog-cloud environments to improve Quality of Service (QoS) in IoT applications. It investigates the optimal distribution of a Service Function Chain (SFC), the building blocks of an IoT application, across the fog-cloud infrastructure, taking into account the intricate nature of IoT and fog-cloud environments.

The primary objectives are to define a platform architecture capable of operating IoT applications efficiently and to model the placement problem comprehensively. These objectives involve detailing the infrastructure's current state, execution requirements, and deployment goals to enable adaptive system management.

The research proposes optimal placement methods for IoT applications, aiming to reduce execution time, enhance dependability, and minimise operation costs. It introduces an approach to effectively manage trade-offs through the measurement and analysis of QoS metrics and requires the implementation of specialised scheduling and placement strategies. These strategies employ concurrency to accelerate the planning process and reduce latency, underscoring the need for an algorithm that best corresponds to the specific requirements of the IoT application domain.

The study's methodology begins with a comprehensive literature review in the area of IoT application deployment in hybrid fog-cloud environments. The insights gained inform the development of novel solutions that address the identified limitations, ensuring the proposal of robust and efficient solutions.

Contents

List of Publications

- Osama Almurshed, Omer Rana, and Kyle Chard. Greedy nominator heuristic: Virtual function placement on fog resources. *Concurrency and Computation: Practice and Experience*, 2021
- Osama Almurshed, Panos Patros, Victoria Huang, Michael Mayo, Melanie Ooi, Ryan Chard, Kyle Chard, Omer Rana, Harshaan Nagra, Matt Baughman, et al. Adaptive edge-cloud environments for rural ai. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 74-83. IEEE, 2022
- Osama Almurshed, Omer Rana, Yinhao Li, Rajiv Ranjan, Devki Nandan Jha, Pankesh Patel, Prem Prakash Jayaraman, and Schahram Dustdar. A fault tolerant workflow composition and deployment automation iot framework in a multi cloud edge environment. *IEEE Internet Computing*, 2021

In this thesis, we have presented only Osama Almurshed's contributions towards developing solutions, conducting experiments, designing adaptive platforms, optimising algorithms, and distributing AI applications in the domain of IoT. Collaborative work not attributed solely to Osama has not been included in this thesis. The other authors have generally contributed by offering initial applications, assisting in debugging and configuration, providing datasets, and helping with manuscript preparation for the published paper.

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ACO Ant colony Optimisation

AHP Analytic Hierarchy Process

AWS Amason Web Service

CPU Central Processing Unit

CSP Constraint Satisfaction Problem

DP Dynamic Programming

EO-GNH Enhanced Optimised Greedy Nominator Heuristic

FL Fuzzy Logic

FN Fog Node

GA Genetic Algorithm

GCP Google Cloud Platform

GDE3 Generalised Differential Evolution

GNH Greedy Nominator Heuristic

GPS Global Positioning System

GPU Graphics Processing Unit

GSA Gravitational Search Algorithm

HYPE Hypervolume Estimation Algorithm for Multi-Objective Optimisation

IBEA Indicator Based Evolutionary Algorithms

ILP Integer Linear Programming

IoT Internet of Things

KH Krill Herd Algorithm

LO Lyapunov Optimisation

LS Local Search

MA Memetic Algorithm

MOCell Multi-Objective A Cellular Genetic Algorithm

MPA Marine Predators Algorithm

MTBF-clock Mean Time Between Failures Clock

MTBF Mean Time Between Failures

MTTF Mean Time to Failure

MTTR Mean Time to Recover

NFV Network Functions virtualisation

NLP Natural Language Processing

NN Next Neighbourhood Search

NSGA-III Non-Dominated Sorting Genetic Algorithm III

NSGA-II Non-Dominated Sorting Genetic Algorithm II

OMOPSO Optimised Multi-Objective Particle Swarm Optimisation

OOAD Object-Oriented Analysis and Design

PC Personal Computer

PSO Particle Swarm Optimisation

PeSOA Penguins Search Optimisation Algorithm

RAM Central Random Access Memory

RB Rule-Based Algorithm

RPi Raspberry Pi

RP Random Placement Replicated Random Placements

RR Round Robin

Rand Random Placement

SA Simulated Annealing

SBC Single-board computer

SDN Software-Defined Networking

SE Search Economics

SFC Service Function chaining

SMPSO Speed-Constrained Multi-objective Particle Swarm Optimisation

SPEA2 Strength Pareto Evolutionary Algorithm 2

VM Virtual Machine

VNF Virtual Network Function

VNS Variable Neighbourhood Search

WOG whale Optimisation Algorithm

ZDT1 Zitzler-Deb-Thiele

To my sons, Ahmed and Abdullah, I hope my academic pursuits have not weighed heavily on your beautiful souls. Thanks to you and your mother for your patience.

Acknowledgements

With heartfelt gratitude for the blessings bestowed upon me by God throughout my academic journey, I extend sincere thanks to the individuals and organisations who significantly contributed to my achievements during my PhD.

First and foremost, I am immensely thankful to my supervisor, Omer Rana, for his unwavering support, guidance, and the freedom he granted me to explore my ideas. Even when I enthusiastically ventured beyond the boundaries of this freedom. Omer's patience and mentorship have played a pivotal role in shaping my research endeavors. Grateful for the Saudi Arabian government's support enabling my research, and to Prince Sattam bin Abdulaziz University.

I am indebted to Kyle Chard, Souham Meshoul, Abdullah Aljumah, Salman Alotaibi and Philipp Reinecke for their invaluable advice and insightful comments. Their advice has significantly facilitated my journey towards a PhD.

I am grateful to the Parsl team at Argonne National Laboratory and The University of Chicago for their assistance in overcoming research challenges, leading to meaningful results.

Special recognition goes to Yinhao Li, Panos Patros, Matt Baughman, Harshaan Nagra, and Ioan Petri for their exceptional assistance in the development of distributed AI applications. I would also like to express my gratitude to Rajiv Ranjan, Devki Nandan Jha, Pankesh Patel, Prem Prakash Jayaraman, Schahram Dustdar, Victoria Huang, Michael Mayo, Melanie Ooi, Ryan Chard, Ian Foster, Chris Anderson, and Stephen Burroughs for their valuable contributions and support.

I am truly blessed to have a supportive family. My parents, Ahmed and Nawal, have always believed in me and encouraged me to aim high. To my wife, Norah Huzaim, I am eternally grateful for her unwavering support, compassion, and meticulous proofreading of my writing. I am immensely proud of my beloved sons, Ahmed and Abdullah, and I extend my heartfelt wishes for their bright and prosperous future. I would also like to express my gratitude to my siblings, Ruba, Saleh, Amr, Khloud, Abdullah, Haneen and Moath, for their constant support and inspiration, especially in the last year of my PhD.

I sincerely appreciate the support and feedback from Ashish Kaushal, Osama Almoghamis and Wafi Bedewi. I am also thankful to my friends and PhD colleagues, Asmail Muftah, Fahd Alhamazani, Turki Al Lelah, Fahad Alodhyani, and Mohammed Asiri, for their shared journey and mutual support during the challenging times of the COVID-19 pandemic. Also, I extend my gratitude to the COMSC-PGR team, particularly Helen Williams and Adam Hammond, for their assistance.

I am profoundly grateful to all mentioned for your contributions and support.

Chapter 1

Introduction

1.1 Motivation

Despite its benefits, the cloud computing model often struggles to fulfil two crucial requirements of many IoT applications: reduced latency and improved data privacy. This issue arises from the time delay associated with data transfer from IoT devices to the cloud, which risks data becoming outdated [1]. Also, transmitting sensitive data to remote cloud servers can raise privacy concerns. Fog computing addresses these issues by extending the cloud model to process time-sensitive data at the network's edge, accelerating application execution while also reducing the need for remote data transmission, thus enhancing privacy protection.

The merging of fog and cloud models into a *hybrid fog-cloud* architecture yields an effective solution. This framework combines rapid application execution with increased computing capacity, providing IoT applications with efficiency and power. These advantages encompass reduced latency, enhanced reliability, and improved Quality of Service (QoS) [2].

Figure 1.1 provides a depiction of the computational offloading process within the IoT ecosystem, composed of three principal layers: the application, platform, and infrastructure layers. The *application layer* incorporates a service functions chain (SFC) [3], which is essentially a group of sub-applications distributed for execution across the fog-cloud infrastructure. Usually, users construct their workflow SFC as

a directed acyclic graph (DAG), wherein the vertices denote functions and the arcs represent data flow.

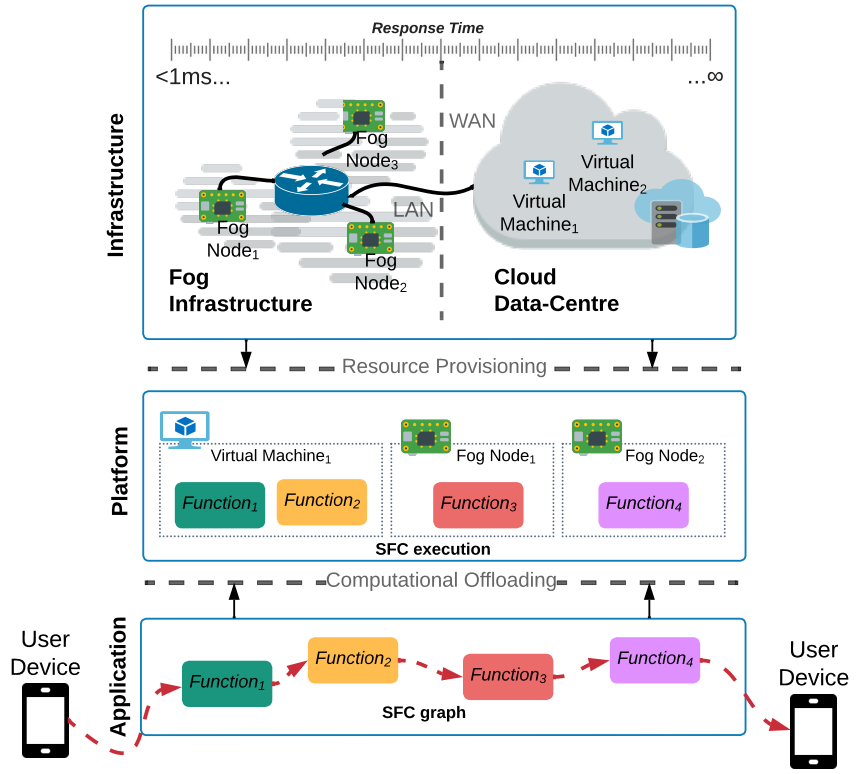


Figure 1.1: The process of IoT computational offloading across various layers, including application, platform, and infrastructure layers .

The fundamental layers draw parallels with the structure of Network Functions Virtualisation (NFV), which uses virtualisation to transform network functions into interconnected virtual building blocks, enabling flexible and efficient service delivery. NFV decouples hardware and software, shifting task management to orchestrator layers (platform), improving flexibility and efficiency, and enabling application logic via application and infrastructure orchestration [4]. The *infrastructure* is the hardware foundation, the *platform* orchestrates service functions, and the *application* represents the software composition of SFC workflow.

The *platform layer*, which bridges the application and infrastructure, is responsible for the execution of service functions within the process nodes of the *infrastructure*. The

designation of these nodes depends on the infrastructure: in fog, they are referred to as fog nodes (FNs), and in the cloud, they are termed virtual machines (VMs). VMs function within cloud datacenters that are accessed over the internet via a wide-area network (WAN). Conversely, FNs are single-board computers (SBCs) situated within a local area network (LAN). These SBCs incorporate various components, including a central processing unit (CPU), random access memory (RAM), and sometimes a graphics processing unit (GPU), serving as nearby computing resources for users.

The *scheduling* mechanism of the platform is charged with establishing the timing and location of execution. Nevertheless, determining the ideal execution location or *placement* for service functions is a complex task. It demands a meticulous evaluation of the process node states, application QoS, and potential bottlenecks to preclude possible delays in IoT applications.

IoT devices fall into two categories: those that collect data via sensors and those that act through actuators. The scheduler, which have been the primary focus of this thesis, do not operate on IoT devices; instead, they perform task execution at higher levels on edge/fog nodes and on the cloud, which process data from IoT devices and can produce actionable outputs.

The scheduler should be designed to manage the unique demands of IoT environments, focusing on meeting low latency requirements to ensure timely responsiveness to environmental and sensor data. This scheduler also adjusts to the varied and constrained network conditions typical in IoT settings, ensuring efficient data communication. It operates effectively in a distributed and decentralised system, coordinating tasks among numerous devices. Moreover, it is highly adaptable to the dynamic nature of the environments, ensuring the system's resilience and operational efficiency despite changing device availability and network conditions.

IoT devices often have restricted computing capabilities and are mostly used for data collection and action execution through sensors and actuators. To overcome such limitations, computational offloading and data migration to more capable computing

resources can be the solution. Fog-cloud infrastructure leverages the computational power of offloading tasks from IoT devices to more powerful process nodes, such as fog nodes or cloud instances, to reduce the computational load on the IoT devices. This Task offloading maintains the energy of IoT devices and ensures data collection for longer duration while complex processes are handled by much more powerful nodes.

However, while offloading tasks can enhance computational efficiency, it may also increase latency and pose privacy risks. Therefore, there is need for the scheduling process that considers latency, risk of data leakage, fault tolerance, network connectivity, and proper utilisation of resources during the task execution process.

Therefore, a pressing need exists for an advanced scheduling algorithm that concurrently considers the requirements of the IoT application and the status of the infrastructure. This algorithm should equip the platform with the ability to adapt to changes in the environment, such as partial system failure, ensuring the avoidance of additional delay. Importantly, the execution time of this algorithm should not add substantially to the end-to-end latency of the SFC execution process.

The subsequent sections of this chapter are organised as follows: The Section 1.2 addresses the challenges associated with managing IoT application management. Subsequently, Section 1.3 emphasises the research problems and the thesis' objective. Section 1.4 describes the research methodology, while Section 1.5 summarises the thesis' contributions and outlines its organisation.

1.2 The Challenges of Managing IoT Applications

The IoT application deployment process in fog-cloud infrastructure faces challenges in each ecosystem layer: infrastructure, platform, and application. For instance, infrastructure components are spread out geographically, and applications have distinct characteristics. These difficulties, which do not occur in traditional systems, make it challenging for the platform to manage IoT applications. This includes the scheduling

challenges encountered in the platform layer. In the following sections, we address these and other challenges at every layer that affect the underlying IoT applications.

1.2.1 Challenges in Application Layer

Application requirements are described as QoS, such as fast response and low cost (i.e., reduce the number of resource utilisations for cost-effective operations). Whereas function requirements are things that govern each function's execution, such as data privacy, a complete deadline, hardware acceleration, software package, etc.

Application requirements in the IoT ecosystem have many aspects and can often be characterised by various QoS metrics. Among these, fast response time and cost-effectiveness are highly important. Fast response time ensures that IoT applications can meet the real-time needs of users, a critical aspect in scenarios like emergency response systems or industrial automation. On the other hand, cost-effectiveness is another comprehensive metric that goes beyond mere financial expenditures. It includes not only the monetary costs associated with deploying and maintaining IoT solutions but also the resource utilisation costs, including energy consumption, bandwidth usage, and other indirect costs related to system maintenance and scalability.

However, function requirements consider the specifics of each service function within the IoT application. These requirements are not just about ensuring the functional integrity of the service but also about adhering to specific operational parameters. This includes ensuring data privacy, a critical aspect as data breaches can have a significant impact on performance. It also involves ensuring complete deadlines (crucial for time-sensitive applications), leveraging hardware acceleration for computationally intensive tasks, and compatibility with necessary software packages, among other factors.

The design of an application's SFC graph is usually separate from the infrastructure that runs it. Infrastructure and application decoupling transfer the execution task to the platform [5]. Therefore, it is important to design the functions of the SFC in a way that

allows the user to assign requirements for each function so the platform can handle them.

The first challenge in the application layer is the design of the SFC graph, which divides the programme logic of the application into functions, each with its own requirements.

The second challenge is enabling parallel programmes and executing the functions' code on a specific process node, i.e., the act of function placement. This entails building a dynamic graph that demonstrates data transformations caused by function execution at particular process nodes.

1.2.2 Challenges in Infrastructure Layer

Infrastructure for IoT applications is geo-distributed by nature, which means that process nodes are in different locations and are linked together through wired and wireless networking. Also, the infrastructure is shared among applications, which means multiple service functions of one or more applications run on the same infrastructure.

Nodes and networks may be unreliable, and therefore functions may fail or be unable to meet the applications' QoS. Also, infrastructure can dynamically change when process nodes are added or removed. This makes it costly to search for a node that provides a service, as the activity of nodes and connections must be continuously monitored.

The first challenge in this layer is to keep track of the infrastructure changes in a reasonable amount of time to avoid increasing latency. The second challenge is to measure unreliability and locate node failures to avoid placing functions on unreliable infrastructure components until they recover.

1.2.3 Challenges in Platform Layer

The platform layer is the intermediate layer that provides resources to the applications. Thus, it should be aware of the system as a whole and adapt to changes that affect its

performance [6].

There are two types of resources: *utility* and *application*. The utility's resources are used to analyse, configure, optimise, and maintain the infrastructure. Whereas the application resources are the resources that the platform offloads applications to.

The adaptation process relies on utility resources to monitor application resources in order to be aware of their state, and then plans a schedule to decide when and where applications are offloaded.

The first challenge for the platform layer is to set up mechanisms for analysis that make the platform aware without overloading the system's resources. These mechanisms have to provide comprehensive information that supports the scheduling process.

The second challenge is determining where the adaptation process occurs, which is dependent on the type of system control. There are either centralised or decentralised types of control [7]. The characteristics of the applications govern the control type. Unreliable wireless networks, for example, require control at the edge device, resulting in a decentralised system. This setup enhances system resilience and response time by enabling localised decision-making and processing.

In a centralised setup, a singular control unit is utilised for the decision-making process. The control node is responsible for analysing the comprehensive data collected throughout the network, making strategic decisions about task allocation and resource management, and assigning tasks to the respective nodes. While this model benefits from a unified decision-making approach, ensuring consistency and potentially simpler management, it also raises concerns about single points of failure and the scalability of the framework. On the other hand, a decentralised control framework distributes the decision-making process across multiple nodes within the network. This model is particularly advantageous in scenarios where network reliability is a concern, such as in environments with unstable wireless connections. This approach is scalable and better at handling failures, but the distributed nature of the network makes it more complex

and difficult to utilise within the network.

The third challenge is to integrate utility tools for distributed programming, infrastructure analysts, and scheduling algorithms to execute IoT applications. This includes integrating the system with other platforms, such as container technology and data streaming tools [8].

1.2.4 Challenges in Scheduling Algorithm

The scheduling process is the task of providing a plan that has the time and place to run an application. It decides placement based on infrastructure monitoring data and the application's QoS requirements. Monitoring data are not only useful for having an initial schedule placement plan but also help in revising the plan in case of infrastructural changes [5].

QoS requirements necessitate the fastest application execution at the lowest possible cost and avoid resources that increase application execution time. This also necessitates the scheduling algorithm to use utility resources at full capacity to accelerate the scheduling process.

The first challenge in scheduling is integrating the scheduling algorithm with the platform and using comprehensive infrastructure information. This also involves determining when the placement decision should be made and how frequently it should be revised.

The second challenge is to evaluate the quality of the scheduled placement plan in light of the conflict between QoS criteria such as performance and cost. For example, increasing the number of process nodes used to execute an application improves reliability and performance but raises the cost.

The third challenge is finding a way to give the scheduling algorithm full access to resources. The access should define the execution configuration, which includes par-

allel vs. sequential runs, synchronous vs. asynchronous execution, shared-memory vs. cluster computing, etc.

The fourth challenge is to find a scheduling algorithm that suits the infrastructure and application characteristics. The scheduling algorithm could perform better in certain scenarios than others. This is affected by various factors, such as the service functions hardware requirements and the type of process nodes and the way they are connected. It is challenging to figure out how to match the best scheduling algorithm to a particular application.

1.3 Research Problems and Objectives

This thesis focuses on the execution of IoT applications on fog-cloud infrastructure with the goal of improving multi-QoS metrics. To address the aforementioned challenges, we aim to achieve the following objectives:

How might we enhance resilience in IoT application placement through a design process that considers infrastructure state, application requirements, and deployment objectives? This question addresses the need for a design process that effectively incorporates infrastructure state, application needs, and deployment objectives for resilient IoT application placement. We aim to develop a model that aids in effective scheduling and enhances resilience in application placement. This research question is explored in detail in *Chapter 3*.

Can we develop an IoT platform that is dynamic and can adapt to changes in IoT application requirements and changes in system configuration? This question investigates how to design an adaptable platform architecture that effectively links platform tools with optimisation algorithms for enhanced system management. We focus

on how this architecture could adapt to shifts in application requirements and technological advancements, while also providing methods to evaluate its effectiveness. This research question is discussed in *Chapter 4*.

How can we define a strategy that optimises the placement of IoT applications, balancing multiple QoS metrics within various IoT domains? This question explores strategies to optimise IoT application placement with the balanced consideration of multiple QoS metrics across various domains. Our aim is to define a method that integrates key considerations for placement, manages trade-offs among metrics, and incorporates a dual-metric evaluation framework emphasising both solution quality and speed of convergence. This research question is explored in detail in *Chapters 5, 6, and 7*.

How can we develop scheduling strategies that prioritise execution performance, cost-effective resource utilisation, and promote high resiliency for IoT applications, taking into account domain-specific characteristics and infrastructure configurations? This question aims to identify and apply optimal scheduling strategies for IoT applications based on unique domain characteristics and infrastructure configurations. Execution performance in this context refers to the expected completion time of tasks, cost-effective resource utilisation corresponds to the cost aspect and is specifically concerned with the efficient use of computing resources, and high resiliency is related to managing failure risks. We aim to develop strategies that are adaptable to different application domains, account for application and infrastructure characteristics, and can handle diverse configurations, irrespective of control type. Evaluations of these strategies, presented in *Chapters 6 and 5*, using different IoT applications are discussed in *Chapter 7*.

1.4 Methodology: An Iterative Process

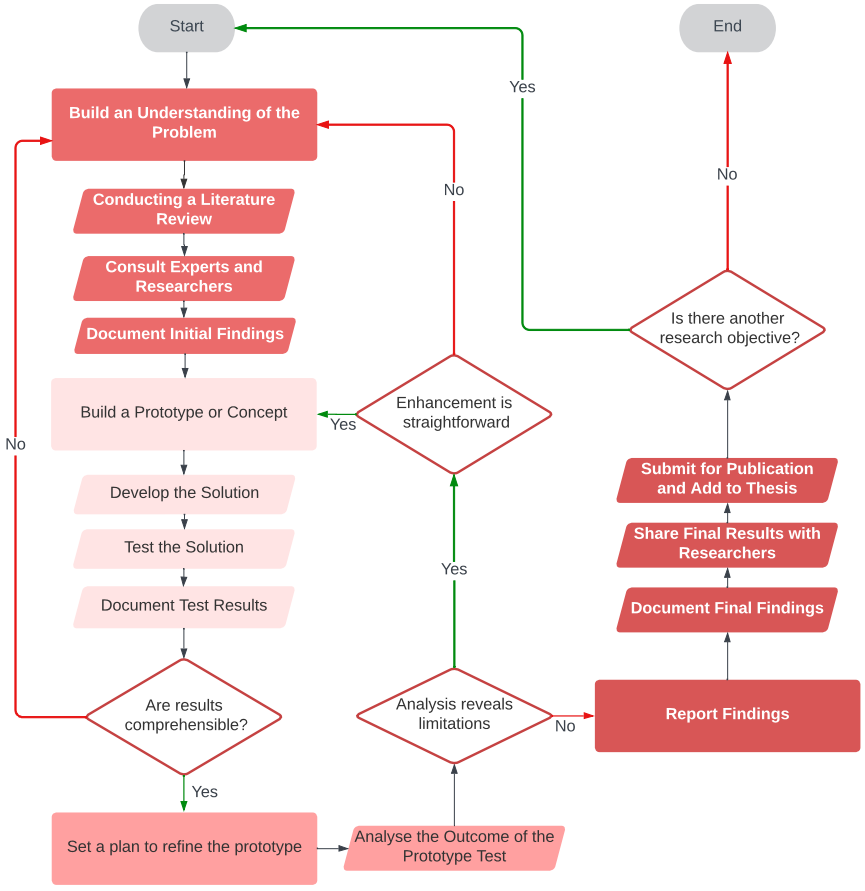


Figure 1.2: This diagram outlines our iterative research methodology, mapping the journey from initial problem comprehension to prototype development, refinement, and final reporting, integrating continuous reviews and expert advice at each phase.

Our research methodology is based on an iterative approach, involving cycles of systematic learning, implementation, and refinement. This cycle continuously incorporates newly acquired information and insights into our work, thereby enhancing our comprehension and enhancing our output.

Important to this strategy is the consistent participation and contribution of other researchers, whose external perspective and expertise significantly improve our work. This methodology, shown in Figure 1.2, supports all phases of our research, from ac-

quiring an understanding of the IoT field to developing applications. In the following example, illustrate how the methodology helped in determining the research.

Understanding the IoT field and operation platforms Our journey begins with an in-depth knowledge of the IoT and its applications. A comprehensive literature review and extensive hands-on experience with the real-time protocol (RTP) and Parsl [9] form the basis of our knowledge in this field. This phase, guided by the supervisor's insights and the assistance of the Parsl team, leads to the essential elements of IoT operation. By integrating replication, environmental or system feedback, our platform provides both system resilience and QoS, enhancing SFC for edge-based applications. Designed to support both real-time and non-real-time applications, our strategy utilises adaptive transmission and precise SFC execution to enhance performance and reliability, effectively addressing the dynamic demands of edge computing. *Chapter 7* highlights our approach's ability to handle real-time and non-real-time applications, emphasising its wide-ranging applicability in edge computing environments.

For example, federated learning updates models with new data through scheduled training tasks, optimising over time—a non-real-time application (Section 7.3). Additionally, the intelligent cooling system reacts quickly to temperature changes to preserve food, exemplifying a soft real-time application (Section 7.4). This knowledge is essential for developing the adaptive platform and simulations that facilitate testing the platform's adaptability (*Chapters 3 and 4*).

Development of an adaptive platform With a solid understanding of the IoT ecosystem, we begin the development of an adaptable platform. This platform integrates monitoring, analysis, deciding, and action (*Chapter 4*). The iterative refinement of this platform, which opens the path for an advanced scheduling algorithm, is motivated by the insights obtained from the comprehensive literature reviews and data from our prototyping and testing with Parsl.

Developing an advanced scheduling algorithm The system's dynamic nature serves as inspiration for our method of developing an effective scheduling algorithm. This results in the development of the Greedy Nominator Heuristic (GNH), which shows initial promise by outperforming Particle Swarm Optimisation (PSO) (*Chapter 5*). To gain insight into the superior performance of our GNH over the PSO, a meta-heuristic method, we build meta-heuristic prototypes and conduct a comprehensive review of the academic literature to clarify the distinctive features of these optimisation algorithms. As we progress and review our method with other researchers, including those from Prince Noura University, a more advanced version, the Enhanced Optimised Greedy Nominator Heuristic (EO-GNH), emerges, validating the good use of the iterative methodology to enhance GNH (*Chapter 6*).

Implementing intelligent IoT application scenarios In the creation and implementation of Intelligent IoT applications, our methodology proves necessary. Each application scenario presents its own set of challenges and learning opportunities, allowing us to test and improve our EO-GNH scheduling algorithm. As we engage with these diverse applications, we become aware of their distinctive characteristics and requirements, thereby refining EO-GNH. This IoT applications, which has been developed with researchers from Newcastle University, the University of Waikato, the University of Chicago, and Cardiff University School of Engineering, enables us to develop robust, intelligent IoT applications that resample scenarios (*Chapter 7*).

1.5 Thesis Contributions and Organisation

This thesis focuses on intelligent IoT application management at the edge of the internet, with a key focus on rapid response time, highly available services, and cost-efficient execution.

The contributions of this thesis are as follows:

- Identify the limitations and challenges of existing adaptive strategies for IoT application frameworks. Analyse and explore how an edge-cloud infrastructure can be utilised to overcome these challenges, *Chapter 2*.

- Modelling the IoT application placement problem in fog-cloud infrastructure, *Chapter 3*. The problem is defined as an SFC placement with replicated functions to ensure the service's availability.

- Simulating the infrastructure and application behaviours so that the platform adaptation mechanisms can be tested in an AI application at the edge-cloud in realistic simulation setting, in *Chapter 4*. The simulation modelling node failure, mobile connection, and application workloads.

- Define a platform for IoT applications deployment in fog-cloud infrastructure, *Chapter 4*. The platform provides the tools required for the system to be aware of environmental changes and adapt to them.

- Developing a parallel optimisation algorithm to solve the placement problem is described in *Chapter 5*. The algorithm makes use of utility resources to speed up the scheduling process.

- Enhancing the distributed scheduling process with state-of-the-art optimisation algorithms, *Chapter 6*. Optimisation algorithms are dynamically selected based on the characteristics of the application and infrastructure.

- Implementing distributed intelligent applications to evaluate the system, *Chapter 7*. The scenarios differ in infrastructure and application characteristics. Also, each application is in a different IoT domain: smart city, smart factory, and precision agriculture.

Figure 1.3 shows the organisation of thesis chapters. *Chapter 2* gives background information and a literature review on the management of IoT applications in a fog-cloud environment. *Chapter 3* models the placement problem and the objectives of the

scheduling process as an optimisation problem. *Chapter 4* defines the system architecture that manages the infrastructure and operates the IoT applications.

Chapter 5 introduces a parallel multi-objective optimisation algorithm. The algorithm utilises the optimisation problem definition in *Chapter 3* and builds on top of the platform in *Chapter 4*. *Chapter 6* enhances the parallel optimisation algorithm with state-of-the-art multi-objective optimisation approaches. The aim is to improve the algorithms' suitability for scheduling IoT applications and adaptation to environmental characteristics.

The proposed adaptive platform and algorithms are evaluated in *Chapter 7* using intelligent IoT application scenarios. The application scenarios differ in their environments and characteristics. *Chapter 8* concludes the thesis with a discussion of the results and future work.

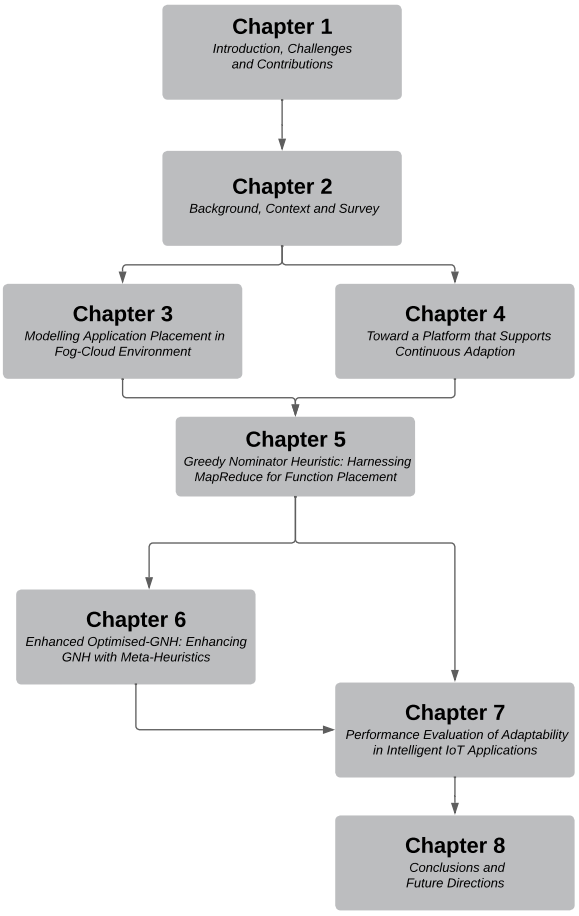


Figure 1.3: Thesis structure

Chapter 2

Background, Context and Survey

2.1 Introduction & Background

This chapter focuses on a literature review of adaptive fog-cloud systems for the IoT application deployment process, as highlighted in *Chapter 1* of this thesis. Understanding the context of IoT applications and fog-cloud computing is crucial to address these challenges.

The combination of the IoT and AI at the edge is creating new opportunities, transforming various sectors with innovative applications. Various applications such as smart homes and healthcare are carefully designed to meet the specific needs of their computing environments. This combination not only makes IoT more extensive but also brings about several complex challenges and opportunities. The main goal is the effective execution of application functions on the infrastructure's process nodes, while ensuring QoS even as conditions change. The coordination of these applications, shown in Figure 2.1, clearly shows the complex relationship between the system's parts and the fog-cloud infrastructure, ensuring that application processes are managed well.

This emphasises the critical need for a resilient mechanism to handle QoS and guarantee continuous functioning, meeting the complex demands of IoT systems with the resources that are presently accessible [10]. It is evident that effectively managing the complex interdependencies and requirements of networks and applications necessitates not only the seamless integration of technical elements but also the implementation of

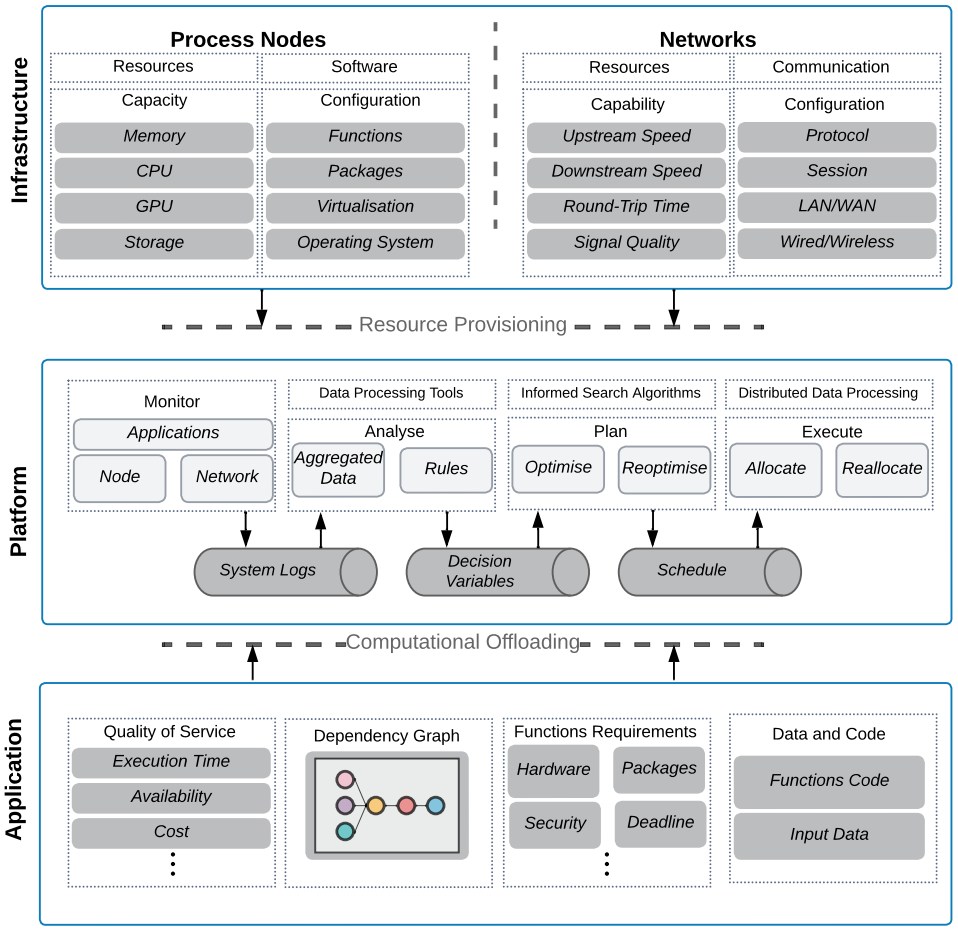


Figure 2.1: Adaptive platform-based application and infrastructure management.

a strategic resource allocation plan and adherence to environmental conditions. By placing emphasis on the development of a flexible framework capable of effectively overseeing QoS, there is a need for an ecosystem that ensures efficiency, capitalising on the complete capability of current infrastructure and resources to provide consistent and superior performance [11].

To effectively manage the complex nature of IoT systems, it is essential to have a methodical approach. This involves regularly monitoring system performance, analysing current statistics, and utilising data streaming tools at the network's edge. These tools have evolved to meet the unique needs of IoT by processing specific data more efficiently and reducing delays, thus enhancing the system's overall performance.

The foundation of this approach is a robust computing framework, ensured by carefully configured nodes and networks. This setup guarantees smooth communication from the network's edge to the node, a critical factor for system reliability.

At the heart of this ecosystem is the platform layer. It orchestrates operations along with the application layer's sophisticated needs and the intricate data dependencies of its components. By optimising the use of node resources, it fine-tunes system execution and quality of service. This orchestration ensures seamless operations across different sectors of the ecosystem, making it well coordinated between the infrastructure component and the application layer.

Building upon the foundational aspects of IoT systems, adaptive systems and mechanisms, like the MAPE-K (Monitor, Analyse, Plan, Execute over a shared Knowledge base) loop, play pivotal roles in enhancing system resilience. These systems are not just reactive but proactive, constantly monitoring the environment and the system's performance [12]. They analyse this data to understand and predict trends, plan responses to these insights, and then execute these plans, all while updating a shared knowledge base. This ensures that IoT platforms are not static but dynamic entities, capable of efficiently adapting to changes in infrastructure, applications, and the surrounding environment. Such adaptability is crucial, especially given the unpredictable nature of the environments where IoT systems often operate.

The MAPE-K loop [12], as an adaptive mechanism, utilises data generated at edge layer to guarantee the smooth operation of applications within the given infrastructure. This loop, embodying the principles of optimisation theory, meticulously analyses the gathered data to formulate and execute well-informed plans, ensuring the application's efficiency and resilience. In this context, data streaming tools are indispensable, facilitating the seamless distribution and management of data across the infrastructure. These tools not only enhance the system's responsiveness but also contribute to the overall robustness of the IoT ecosystem.

For detailed insights into the optimisation strategies and data streaming engines' func-

tionalities, refer *Appandex A* to the appendix of this thesis. The appendix provides an overview of the various optimisation approaches and a concise introduction to data streaming, illustrating their significance in the ecosystems.

This chapter is designed with an aim to provide a review. Categorisation and analysis of existing Approaches and critical evaluation of Approaches are pivotal in understanding the landscape of current research (*Section 2.2*). Then, we will examine the strategies employed by researchers in this field to get solutions (*Section 2.3)*. We conclude this chapter with identification of challenges and directions of this thesis, providing a clear trajectory for future research (*Section 2.4*). Additionally, we have provided more details in *Appendix A*, *Appendix B* and *Appendix C* about our approach, protocol, and tools utilised for conducting this review.

2.2 Categorisation and Analysis of Existing Approaches

2.2.1 IoT Application Domains

Diverse sectors are harnessing the power of connectivity and data. Smart homes integrate devices for remote management of home features, offering convenience and savings. Smart cities deploy technologies for enhanced urban living, tackling challenges like pollution and traffic [13]. Industrial IoT leverages data from machinery for insightful business decisions, boosting efficiency [14]. Smart Healthcare utilises technology for patient monitoring and hospital management [15]. Precision agriculture employs sensors and drones for resource-efficient farming [16]. Lastly, smart vehicles incorporate technology for improved transport services and automation. Each sector, uniquely benefiting from IoT, signifies a transformative step towards a more interconnected, efficient world [16].

2.2.2 Optimisation theory & Optimisation Attributes

Application objectives in IoT systems span various key performance indicators. Service availability ensures system resilience against disruptions, maintaining uptime. Completion Time focuses on efficient resource and schedule management to expedite task execution. Deployment costs encompass both explicit and implicit financial aspects of service usage and maintenance. Energy consumption prioritises efficiency in resource usage and optimised functions for sustainability. Data security involves stringent measures to safeguard information and restrict access. The service scale dynamically aligns resources with request volume. Application's throughput measures the system's efficiency in processing workloads over time, influenced by performance metrics and network factors. These objectives, identified from our observations, set the stage for an in-depth exploration and optimisation of system performance, a process briefly overviewed in Section 2.3 and elaborated in detail in the Appendix A.

Efficient scheduling is essential for various applications and systems, involving organising tasks and allocating resources to ensure optimal utilisation and timely completion. It involves searching for scheduling solutions out of a set of possible solutions known as the search space [17].

The search space is the set of all possible solutions to a problem, and optimisation algorithms aim to identify the best solution within this space. The search space can be vast and complex, posing challenges to the optimisation process. Informed search algorithms play a pivotal role in enhancing the scheduling of tasks, leading to better performance.

Informed search explores the search space using heuristic functions and objective functions to gauge and evaluate their quality based on specific criteria. A heuristic function is typically employed to drive the search process toward the global optimum, but it will almost always result in a good approximation [18]. This improves efficiency in finding the best or near-best solution.

The structure of an algorithm is defined by several key components that contribute to its effectiveness and efficiency. Objective functions and constraints define the problem and the algorithm that optimises the solution. The objective functions are the central aspect, representing the function that the algorithm aims to optimise by either minimising or maximising its value. Constraints are also essential, imposing conditions or restrictions that candidate solutions must satisfy to be considered feasible.

Algorithm components are essential steps in optimisation algorithms that guide the search process towards optimal or near-optimal solutions. They involve selection and variation mechanisms. Selection prioritises promising solutions based on their quality, while variation generates new candidate solutions by modifying selected previous solutions [19]. The combination of these procedures ensures a balance between maintaining promising solutions and exploring the search space to avoid local optima and converge to global optima (balance between exploring and exploiting). Heuristics play a crucial role in this structure, as they optimise solutions for complex problems by utilising domain knowledge and insights to create rules or guidelines that guide the search towards promising areas in the search space. Integrating heuristics into the design of an algorithm enhances the optimisation process's effectiveness and efficiency.

The properties of multi-objectives optimisation algorithms significantly impact the quality of solutions, making it essential to consider them when designing optimisation methods. Figure A.4 in Appendix A.4.3 summarises the properties.

These properties include preference information, which involves the various types of preference data provided by a user or decision-maker to the optimisation's goals, such as a priori, progressive, or a posteriori preference information, or even no articulated preference information [20].

Solution evaluation encompasses methods used to assess and compare the quality of solutions within the search space, employing techniques like scalarisation and Pareto optimality [21]. In multi-objective optimisation, scalarisation aggregates multiple objective functions into a single one, while Pareto methods prioritise non-dominated solu-

tions across all criteria among other explored alternatives. A set of non-dominant solutions offers a variety of potential alternatives, each with its own unique trade-offs; none of them is better than the other in all objectives [21].

Local and global optima are two important concepts in the field of optimisation. Local optima refer to the optimal solutions within a limited area or region of the search space, while global optima refer to the optimal solutions throughout the entire search space. Noting that a local optimum is not necessarily a global optimum is essential, as other regions of the search space may contain better solutions [17].

Exploration and exploitation are fundamental to the optimisation process. Exploration is the process of discovering new regions of the search space, which may provide global optimums by revealing diverse solutions. Exploitation, on the other hand, entails augmenting the currently best-known solutions and refining them to discover local optima within a particular region of the search space. It is essential to find a balance between these two strategies; excessive exploitation could cause the algorithm to become trapped in suboptimal solutions, while excessive exploration could prevent the algorithm from adequately refining promising solutions. Therefore, a successful optimisation procedure must navigate carefully between exploration and exploitation [17].

The execution approach pertains to how the optimisation algorithm is carried out, incorporating parallelism and serial execution, where parallel algorithms can accelerate the search process while serial algorithms execute tasks sequentially [22].

Solutions quantity outlines methods used to navigate the search space, categorising optimisation algorithms based on the number of concurrent solutions, including single solution-based algorithms (trajectory methods) and population-based algorithms that operate on a set of solutions simultaneously [19].

2.2.3 Objectives Functions and Constraints in Optimisation Algorithms

Several critical factors contribute to the overall effectiveness and efficiency of operations. *Service availability* ensures uninterrupted operation despite challenges like network or hardware failures. *Completion time* optimises resource use to expedite project execution efficiently. *Deployment cost* covers both direct expenses and ongoing maintenance, emphasising resource use efficiency. *Energy consumption* is minimised through optimised resources and energy-efficient programming. *Data security* safeguards integrity and restricts access through methods like encryption and access controls. *Service scale* dynamically adjusts to application demands, optimising resource distribution. *Application throughput* measures workload processing speed, focusing on enhancing efficiency through performance and latency improvements. In multi-objective optimisation, a model constraint can address particular objectives, such as reducing node overload [23] or delay via deadline constraints [24, 25, 26]. To avoid constraint violations, it is important to recognise that delay control can serve as a constraint, not an objective. Solutions addressing an objective as a constraint are less likely to be optimal, particularly if they are independent of the objective function, which does not use it directly during evaluation and acts as a filtering mechanism.

Sometimes delay is dealt with as a priority problem, which allows the most urgent tasks to be run before other tasks in the queue. This prevents high waiting times for latency-sensitive tasks [27, 28].

2.2.4 Architectural Aspects Impacting the Adaptive Loop

Exploring the architectural aspects affecting the adaptive loop highlights the cooperation between various components and strategies. In addition to the optimisation algorithm, several studies describe the integration of adaptive measures, such as replication, as an important approach for enhancing system resilience and performance [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]. In edge computing, blockchain emerges as a robust mechanism for tracking and rewarding node contributions, ensuring the accuracy and authenticity of processed tasks [41]. Moreover, security tags serve as

a safeguard, maintaining the confidentiality of IoT devices by harmonising execution performance with a trust-centric IoT service placement approach [42].

Enhancing system adaptability often involves incorporating additional components. For instance, the integration of a data-aware module can significantly refine data management and processing capabilities. This enhancement not only elevates the system's proficiency in handling diverse data types but also optimises query times and resource utilisation [43]. Moreover, adaptability is not confined to structural elements; it also permeates the algorithmic logic. As highlighted in certain studies [44], algorithms such as the modified genetic algorithm can dynamically adjust to IoT application scenarios, striking a balance between energy efficiency and latency, thereby catering to the varied demands of IoT applications.

The concept of *availability* is complex and interpreted differently across various studies. For some, researchers [45], it denotes the capacity to fulfil deadlines, while others, including Wang et al. [46], Gong et al. [47], Mohamadi et al. [34], and Moallemi et al. [48], emphasise on aspects like geographical coverage. Similarly, the notion of *cost* transcends beyond mere expense reduction; it encompasses broader strategies such as profit maximisation, a perspective particularly evident in works like that of Wang et al. [46].

In dynamic settings, certain algorithms inherently possess the capability to address challenges autonomously, a trait discernible through their operational patterns. This innate propensity is often similar to a *greedy approach*, especially in cooperative environments where decision variables undergo continuous refinement. Although not always explicitly labelled as such, this approach mirrors the algorithm's tendency to opt for the most advantageous choice for each task or function, aligning with the essence of a greedy strategy [31, 49, 50, 51, 52, 53, 54, 55, 40]. In this collaborative landscape, decisions are collectively shaped by multiple parameters , such as adapting in real-time to evolving requirements. This collaborative dynamic often employs a form of *parallelism*, understood as cooperative interactions between nodes, thereby

fostering a cohesive decision-making framework often referred to as *joint optimisation* [37, 56, 27].

The focus of the matter lies not within the computation, but within the management of data, particularly its storage and replication. The latency associated with data storage can be decreased by employing replicas, as demonstrated in studies such as Huang et al. [40] and Shao et al. [39].

Certain research even utilises data replicas to investigate the optimal locations for installing these replicas for data-intensive IoT applications within fog computing. iFogStorM addresses data storage delay through multiple data replica placements within Fog computing [28]. Replicas are also leveraged to enhance service availability [34].

In the course of investigating optimisation, we found some studies that have considered the aspect of software architecture. For instance, [38] adopts a digital signature service to maintain data integrity, while [43] implements graph databases. Some research, such as [32], employs system components to monitor backups, utilising both a backup and a standby instance for fault tolerance, supplemented by heuristic algorithms. Other studies, like [43], shift their focus towards the design and specification of a data streaming engine.

2.3 Literature Review's Statistical Analysis

2.3.1 Research Focus Across Various IoT Domains

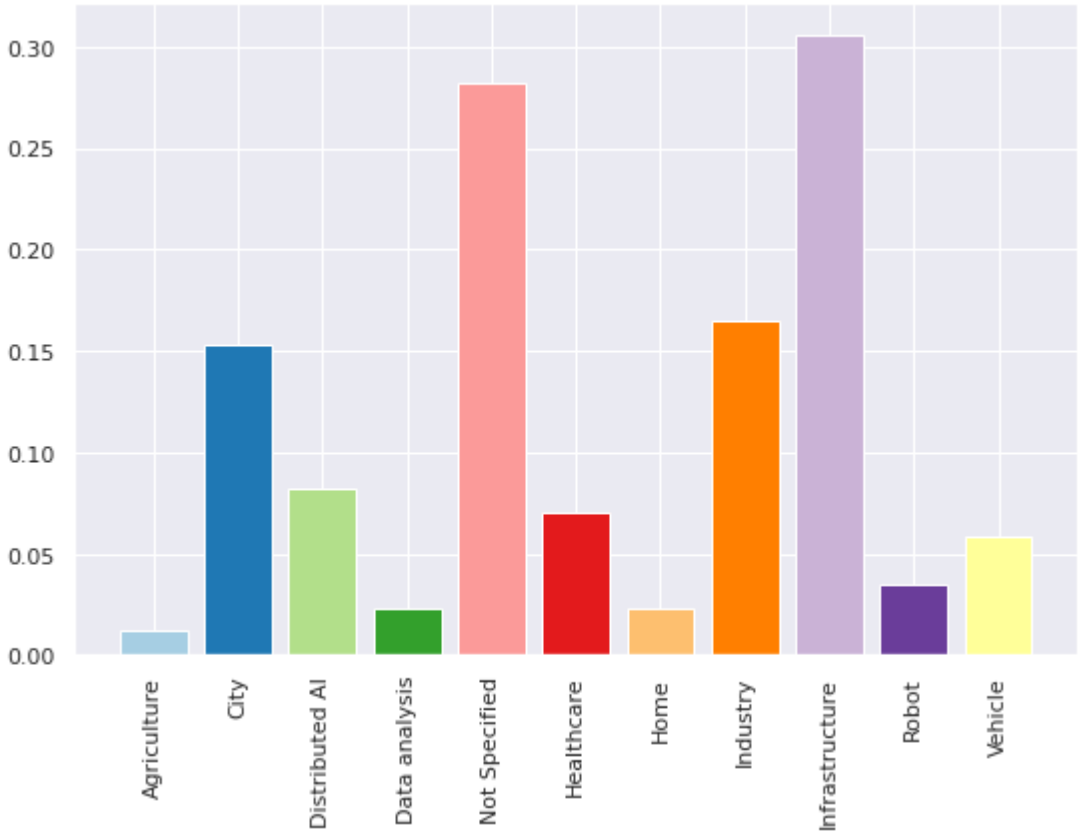


Figure 2.2: The relationship between the IoT domain and the papers is depicted by a bar chart. The percentage represents the proportion of papers in the survey that cover a specific area of study.

Figure 2.2 shows the percentage of research papers that concentrate on various IoT domains. In the context of Network Function Virtualisation (NFV) and Software-defined Networking (SDN), *Infrastructure* management leads with approximately 30.59%, indicating a strong emphasis on research in this domain. This is followed by *unspecified domains* at 28.24%, indicating that a significant number of studies concentrate on generic IoT technologies. Instead of concentrating on specific domains, these studies put more emphasis on the development and improvement of the optimisation algorithms

themselves.

The third-placed *industry* sector accounts for 16.47% of papers, demonstrating its importance in IoT research. Following this are *city* applications, or urban IoT, which account for 15.29%.

Distributed AI and *healthcare*, with applications including machine learning and deep learning, are also well-represented, with 8.24% and 7.06% mentions, respectively, indicating the growing significance of AI in IoT and the transformative potential of IoT technologies in healthcare.

The *vehicle* domain's 5.88% share reflects interest in autonomous vehicles and vehicle-to-vehicle communications, among other topics. Similarly, *Robot*, *Data Analysis*, and *Home* each have 3.53%, 2.35%, and 2.35% representation, respectively. These industries are essential because they encompass expansive fields such as home automation, big data, and robotics, which are integral to the IoT.

Surprisingly *agriculture* has the lowest representation at 1.18%, which may be attributable to a lower adoption rate of IoT technologies in this sector or a decreased focus in academia, despite the potential for a significant impact.

While there is a clear emphasis on infrastructure and industry, the diversity of domains indicates a broad investigation of IoT technologies across a variety of applications. However, the underrepresentation of some domains, such as agriculture, may suggest areas for future investigation.

2.3.2 Distribution of Optimisation Objectives in IoT Research

Figure 2.3 illustrates the proportion of IoT research papers that addressed specific optimisation objectives.

Notably, *completion time* is the leading objective, with approximately 91.76% of papers addressing it. This highlights how important it is in IoT systems, where efficiency

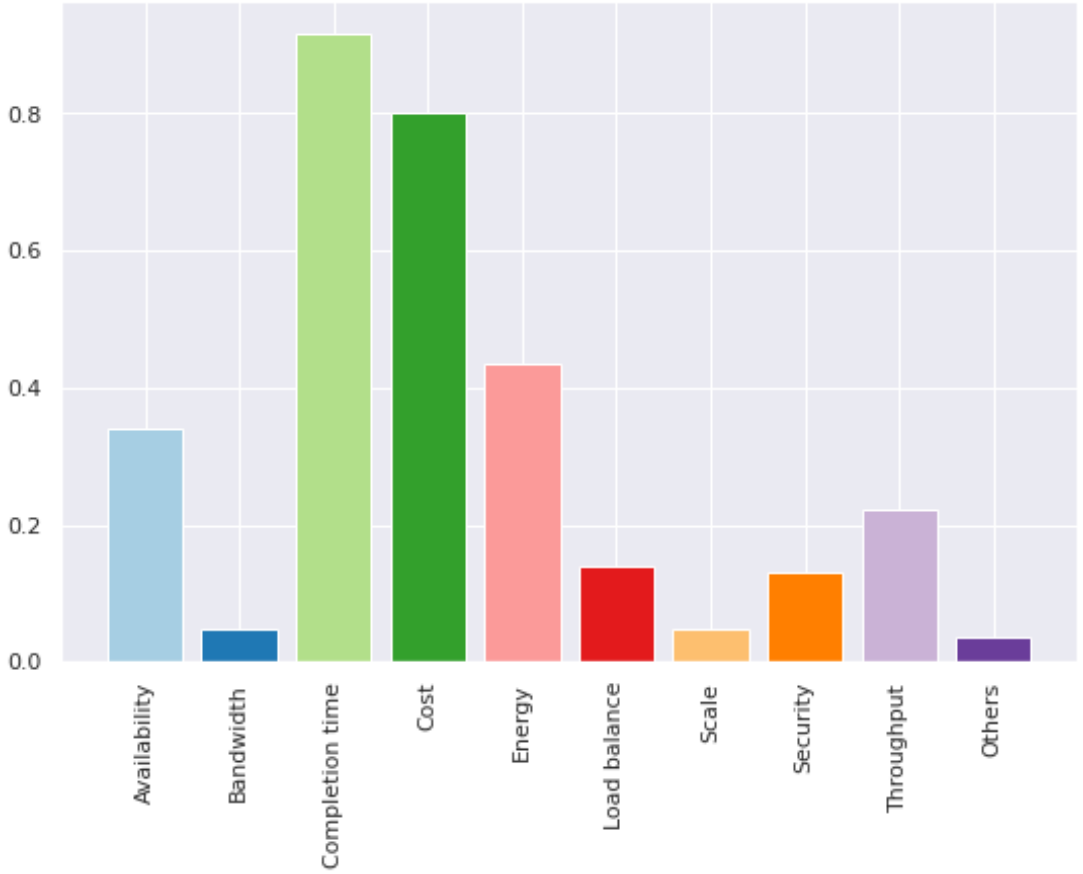


Figure 2.3: A bar graph illustrating the objectives' relationship to the topic of the paper The percentage represents the proportion of papers surveyed that included an explanation of the study's objectives.

and timely completion of tasks are frequently of the highest priority.

Cost is also an important topic, with 80% of papers addressing it. This corresponds with the shared objective of reducing operational costs or increasing cost effectiveness in IoT implementations.

The third most prevalent objective is *energy*, which appears in 43.53% of the papers. This reflects the industry's emphasis on energy efficiency, which is essential in IoT environments where power resources are frequently limited.

Approximately 34.12% of papers address the *availability* objective, reflecting its importance in assuring reliable and consistent service in IoT systems. This is followed

by *throughput* (22,35%), an important measure for IoT systems that often need to process large volumes of data. Throughput is usually measured by the amount of data processed per unit of time.

Load balance (14.12%) and *security* (12.94%) also appear frequently as objectives in the papers. These reflect the emphasis placed on optimising resource utilisation and ensuring data privacy and system security, both of which are crucial aspects of IoT systems.

Bandwidth and *scale* are essential aspects of the IoT, particularly in scenarios involving data-intensive applications or large-scale deployments, despite being less frequently discussed (each in 4.71% of papers).

Others (3.53%) encompasses a variety of objectives not explicitly enumerated, further emphasising the diversity of issues and goals that IoT researchers address.

Overall, the data reveals a wide range of IoT research objectives, with a focus on completion time, cost, and energy efficiency.

2.3.3 Algorithmic Properties Analysis

This part discusses different aspects of algorithmic properties and how each has been handled in the given data set. These aspects range from the nature of the algorithm's process and solution to preferences in multi-objective optimisation and trade-offs between exploration and exploitation. The section also assesses evaluation methods and the types of variables involved.

The optimisation algorithms are characterised by several factors: covering process (parallel or serial), solution quantity (single or population-based), preference (posteriori, progressive, or priori), search (trade-off exploration and exploitation), evaluation (non-Pareto or Pareto), and adaptiveness (static or dynamically reoptimising scheduling plan).

Algorithmic *process* defines the mode of operation of an algorithm, whether it executes steps sequentially (serial process) or simultaneously (parallel process). The majority of the papers (74.12%) used a serial process, while 25.88% opted for a parallel process.

The solution aspect refers to how data is manipulated during the search process, whether it involves multiple entities (population-based) or just one (single). 64.71% of the papers employed population-based solutions while 35.29% used single solutions.

The *algorithm* aspect pertains to the logic underpinning the search program, which is generally connected to a state-of-the-art approach. Among the algorithms, Genetic Algorithm (GA) was used in 43.53% of the papers, followed by Role-Based (RB) in 21.18% of the papers, and Particle Swarm Optimisation (PSO) in 15.29% of the papers.

The *preference* element relates to multi-objective optimisation and when to apply trade-offs between conflicting objectives. 74.12% of the papers used a priori preference, 23.53% used a posteriori preference, while less than 2.35% for a progressive approach. This is assuming that priori and a posteriori in two separate phases act as progressive.

The balance between *exploration* (ER) and *exploitation* (ET) is another important aspect of an algorithm's behaviour. A majority (81.18%) opted for a balanced exploration-exploitation approach, while 16.47% leaned towards an exploitation-oriented strategy. Only about 1.18% preferred an exploration approach (one case).

The *evaluation* aspect refers to the method used to assess trade-offs between objectives. Most of the papers (74.12%) used non-Pareto evaluation methods, while 25.88% used Pareto evaluation.

The *variables type* aspect concerns the frequency of decision variable updates. Does it the algorithm *adapt* to environment changes or not. 68.24% of the papers used static variables while 31.76% used dynamic variables.

2.3.4 Algorithm Application Across Different Domains: A Comparative Analysis

GA: Genetic Algorithm	PSO: Particle Swarm Optimisation	SA: Simulated Annealing
ACO: Ant Colony Optimisation	AHP: Analytic Hierarchy Process	MPA: Marine Predators Algorithm
CSP: Constraint Satisfaction Problem	WOG: whale Optimisation Algorithm	PeSOA: Penguins Search Optimisation
SE: Search Economics	KH: Krill Herd algorithm	GSA: Gravitational Search Algorithm
VNS: Variable Neighbourhood Search	NN: Nearest Neighbourhood Search	RB: Role-Based Algorithm
RR: Round Robin	FL: Fuzzy Logic	DP: Dynamic Programming
MA: Memetic Algorithm	LO: Lyapunov Optimisation	LS : Local Search

Table 2.1: Enumeration of algorithm acronyms in investigated academic articles

This section quantifies the implementation of various algorithms in a variety of domains to demonstrate their adaptability. Table 2.1 presents a list of acronyms for algorithms used in the survey. The emphasis is on the vast number of papers employing these algorithms, which provides crucial guidance for future research and practice. Figure 2.4 displays a scatter plot depicting the diverse algorithm usage across IoT domains. The size of each point indicates the number of publications utilising the respective algorithm in a specific domain.

The wide usage of GA across different fields is notable. GA has been applied in twelve IoT cases, eight industrial cases, and twelve instances involving SDN and NFV. A detailed breakdown of the applications and corresponding references of the GA in different domains is provided in Table C.1 in Appendix C.1.

The wide usage of ACO and PSO algorithms across different fields is notable. PSO algorithm is frequently used, with five industrial cases, three IoT instances, and three infrastructure applications. ACO demonstrates a broad range of applications. Both industry and generic IoT applications (unspecified domain) report their use in two papers each. Table C.2 in Appendix C.1 provides a comprehensive analysis of the applications and references of PSO and ACO in various IoT domain.

Highlighting the unique environment of infrastructure defined by the intricacies of

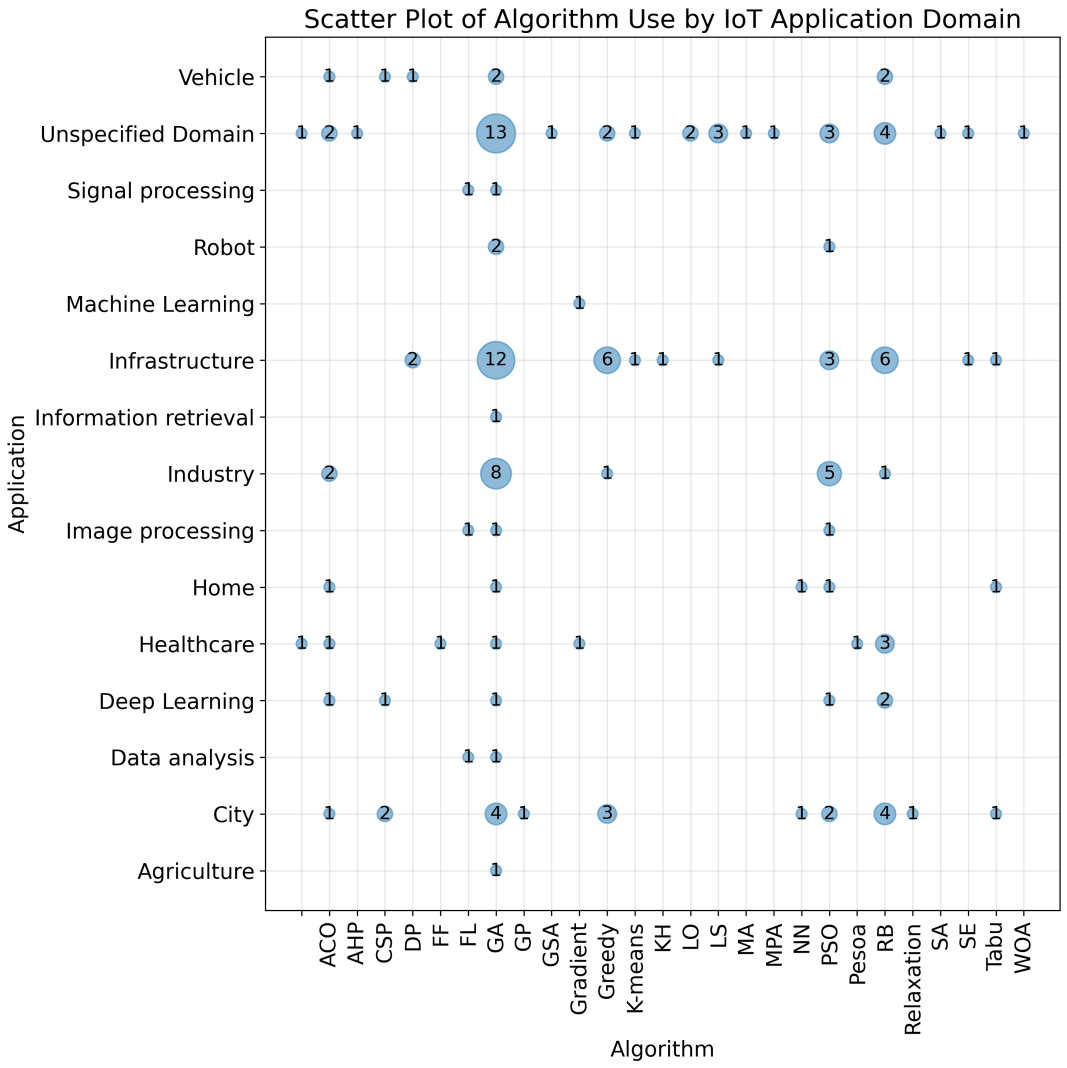


Figure 2.4: Scatter plot representing the use of various algorithms across different IoT domains. Each point's size is indicative of the number of papers utilising the corresponding algorithm for a specific domain.

SDN and NFV, there is a particular focus on joint optimisation techniques and algorithms such as DP [29, 30], which are well-suited for managing the complex dependencies inherent in such environments. Algorithms like Tabu Search [49, 57] are also particularly prevalent in this area and feature in different infrastructure applications. RB algorithms also boast a large range of applications. Six papers implement role-based approaches, and city applications alone feature them in six instances. The algorithms' versatility is further demonstrated by their use across numerous other

domains, potentially due to the simplicity of their application. The applications of the Greedy and RB algorithms in various domains are outlined in Table C.3 in Appendix C.2.

While FL and genetic local search algorithms might not be widely used in IoT, they hold unique roles, with the latter seen as a novelty. The implementation of the FL algorithm across multiple domains such as image processing, signal processing, and data analysis can be found in Wu et al. [58].

The LO algorithm diversifies the spectrum further, given its widespread usage in control theory and electronic engineering [59, 60]. It is noted that it is usual to use greedy algorithms with congestion in DP, considering their step-by-step decision-making property. However, this property is inherent in the programming approach of DP. Notably, LO [59, 60] and DP [29, 30] have each been applied in two infrastructure cases.

Lastly, the k-means algorithm has been used in one IoT scenario [61] and LS in two instances [62, 52], though the specific IoT domain of application remains unspecified.

Table C.4 in Appendix C.2 provides a detailed summary of various algorithms along with their specific applications across different domains. This encompasses the remaining computational strategies not previously discussed, thereby offering a comprehensive overview of algorithm utilisation. Finally, a detailed comparison of optimisation algorithms is presented in Table C.5 in Appendix C.2.

2.4 Open Issues and Positioning

This section highlights the unresolved challenges and gaps within our research field. We position our study in the context of existing work, highlighting our contributions and the potential focus.

The majority of the research predominantly concentrates on the mathematical and theoretical aspects of the deployment issue. It does not extensively address the practical

and engineering components of the optimisation algorithms that function at the network's edge.

They allocate less attention to the time involved in the decision-making process for devising a plan for applications at the edge. This time frame frequently impacts the total completion time, thereby affecting the end-to-end latency. Consequently, it is important to measure the time utilised by the edge device, such as a single-board computer like the Raspberry Pi, as this duration adds to the overall end-to-end latency of the application's execution.

In many studies, applications are often described as intelligent, even if they do not include specific algorithms like those from machine learning. Furthermore, it is important to shift the focus towards AI solutions at the edge, using popular approaches such as federated learning, model training inference, and distributed machine learning. This transformation is important for unlocking the full potential of AI within edge computing frameworks and ensuring its effective management.

The thesis aims to contribute to filling gaps in edge computing and AI, moving from a theoretical focus to practical optimisation algorithms at the network's edge. It seeks to address the decision-making time in edge applications, a factor critical for end-to-end latency and performance. Also, the thesis suggests a shift towards actual AI solutions, utilising approaches such as federated learning and distributed machine learning. This approach is intended to manage AI's potential within edge computing effectively. The goal is to have theoretical insights with practical implementations, aiming to advance the development of robust and intelligent edge computing solutions.

Chapter 3

Modelling Application Placement in Fog-Cloud Environment

3.1 Introduction

In Chapter 2, the foundational context for research on IoT applications and fog-cloud computing is thoroughly presented. The chapter also examines cutting-edge optimisation techniques designed to effectively manage IoT environments and explores existing research limitations. This comprehensive overview provides insights into the state-of-the-art optimisation strategies, while simultaneously highlighting potential areas for research progress. The present chapter introduces the formulation of the placement problem, serving as the initial step towards offering a solution that effectively tackles the existing research limitations

Integer Linear Programming (ILP) is a mathematical optimisation technique that involves a linear objective function and linear constraints, with decision variables constrained to integer values. ILP is particularly suitable for modelling discrete decisions in practical applications, such as scheduling [63]. This chapter presents examples that demonstrate the placement problem in a fog-cloud environment, facilitating a better understanding of the system and guiding the formulation of valid assumptions. Utilising ILP for problem formulation includes defining *decision variables*, *objective functions*, and *constraints* that accurately represent the system and its limitations. Decision vari-

ables are associated with applications and infrastructure data, while objective functions guide the analysis and evaluation of decisions. Logically consistent constraints ensure that the proposed placement solutions are both applicable and realistic.

The chapter focuses on formulating the application deployment problem in the fog-cloud infrastructure using ILP. The placement problem is defined as an SFC placement that involves deploying replicas for each function within the fog-cloud infrastructure. Emphasising the importance of functions occurring early in the SFC, as their successful completion is crucial to preventing downstream failures. These critical functions are assigned higher priority in terms of resilience. Moreover, the problem is framed as a search for resources that optimise *delay*, *risk*, and *cost*, with trade-offs between cost and performance when deploying replica functions.

The remainder of the chapter is structured as follows: Section 3.2 describes an utilisation scenario for the SFC in the cloud-fog ecosystem. Section 3.3 introduces the problem and provides an overview of the system. The system mode is depicted in Section 3.4. The final section concludes this chapter.

3.2 Usage Scenario

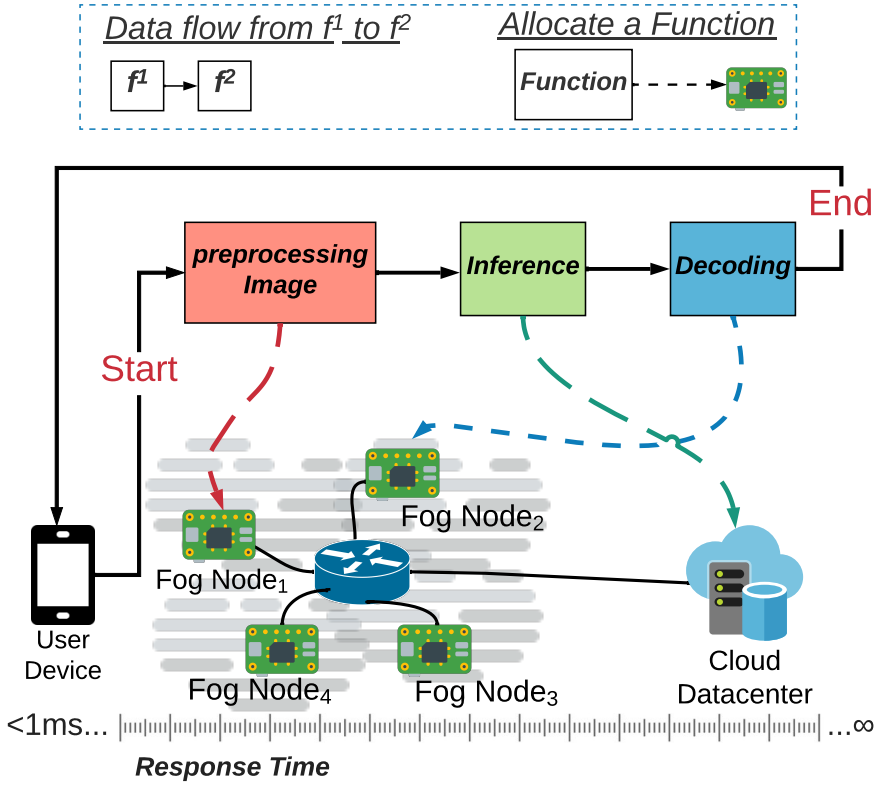


Figure 3.1: Placement of service function chaining graph in fog-cloud infrastructure .

Figure 3.1 depicts a SFC scenario within a fog computing environment. The figure illustrates the execution of data flow within a fog-cloud infrastructure for an AI at the edge application, detailing the operations that carry out inference on an image dataset. The process begins with preprocessing images at the user device level in the fog infrastructure, followed by inference, and then response with inference results in decoding. The functions are allocated and executed across multiple fog nodes (from Fog Node 1 to Fog Node 4) and a cloud datacenter, emphasising the distributed nature of processing in fog computing infrastructure. The sequence showcases the interaction between the user device, fog nodes, and cloud datacenter for task placement in a fog-cloud environment.

As shown in Figure 3.1, the section describes a scenario of application placement in the Fog-Cloud system. The application is divided into three functions: *image preprocessing*, *inference*, and *decoding*. *Decoding* depends on inference and *image preprocessing*, while *inference*, in turn, depends on the *image preprocessing* function. *Image preprocessing* involves enhancing and preparing images for further analytical processing. *Inference* entails deriving insights or making predictions, such as identifying objects in an image, using a trained model. *Decoding* translates the model's output data into a human-readable format that people can easily understand and use.

A user device (for example, a smart phone) may request application functions from a controller Fog Node (FN) in an SFC, and the controller will distributively execute these functions across the infrastructure. Because the *Decoding* is dependent on both the *Inference* and *Preprocessing Image*, the application may experience delays if either the the *Inference* and *Preprocessing Image* fails. As a result, *Inference* and *Preprocessing Image* must have greater redundancy than *Decoding* to reduce redeployment time in the event of a failure. This is based on the observation that functions that occur earlier in an SFC are more important because their failure will have an impact downstream in the SFC pipeline. Greater redundancy at the early stages of the pipeline is likely to provide greater benefits in terms of avoiding delays at later stages of the SFC.

3.3 Problem Statement

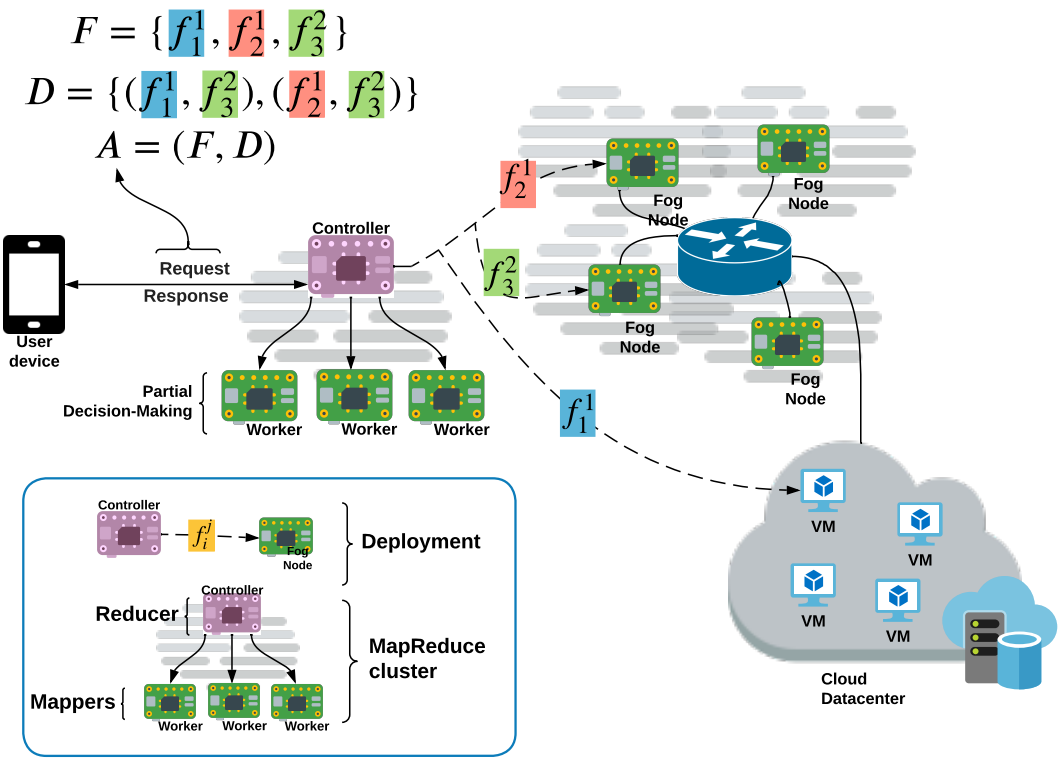


Figure 3.2: Redundant deployment of application A in the fog and cloud. Utilising computer cluster to speed up the analytic and scheduling process.

To reduce latency in real-time applications, fog computing is an intermediate infrastructure between edge devices and cloud systems. An SFC can be used to represent the inter-dependencies between service functions in an application (SFC). It is possible to run SFC service functions from multiple locations. They are prone to failure and frequently fail to meet deadlines. There can be delays, possible data loss, and increased costs as a result of function relocation to other nodes in the system. This delay is also is a result of the time required for the process node to recover from a freeze. This time interval varies from node to node and can be affected by a number of variables, one of which is the programme itself, which demands a large amount of resources to execute functions, and the node is unable to handle the request. Based on redundant deployment and failure tracking of service functions, we design a parallel technique.

Each function is replicated at several locations, taking into consideration the expected completion time, the risk of failure, and the cost.

Failure is defined as a task exceeding its pre-established deadline, rather than simply completing its computation. This deadline is an estimated completion time set before deployment. The impact of failure is thus measured against this timeline.

Locations are processing nodes in our proposed SFC control architecture that are in charge of hosting service functions. Cloud-based virtual machines (VMs) or cloud-based fog nodes are examples of fog nodes (FNs). The *controller* node, which is in charge of managing SFC placement, sends service function requests to locations.

Locations respond to requests by carrying out service functions' execution. Controllers are responsible for managing communications between locations as well as monitoring their capacity, availability, and service functions execution operations. However, updating a controller with the current state of the infrastructure and looking for the best places to deploy functions can be a time-consuming and error-prone process. As a result, the burden of decision-making is distributed among workers, that assist a controller in gathering information about the current state of infrastructure and determining optimal locations, as illustrated in Figure 3.2. In a distributed computing environment, worker nodes can be implemented as virtual instances within the main controller, utilising lightweight containerisation or parallel processing methodologies. Alternatively, these nodes can be embodied by fully-functional standalone computers, such as SBC computers, employing shared-nothing architectures like MapReduce [64]. This is done by using parallel data processing techniques to fully utilise the controller resources (Fully utilising the controller's resources is described in detail in *chapters 4, 5, 6* .).

3.4 System Model

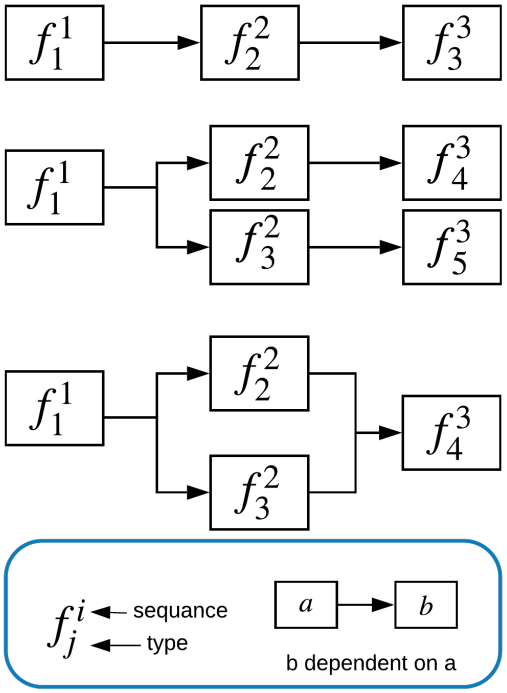


Figure 3.3: Service function chaining examples in abstract graphs; displays input-output data dependencies .

An IoT application is made up of service functions (SFC). SFCs are typically designed as directed acyclic graphs (DAGs), with nodes representing functions and arcs representing data flow direction, and comes in variety of forms, such as Figure 3.3. A DAG workflow with the input data is sent to the system for processing and execution, which is the act of placement (as shown in Figure 3.1).

The controller receives an application (i.e., SFC) as a graph $A = (F, D)$, where F is a set of functions, $F = \{f_1^1, f_2^2 \dots f_j^i\}$, and D is a set of pairs representing dependencies between functions, $D = \{(f_1^1, f_2^1), \dots, (f_{j-1}^{i-1}, f_j^i)\}$. The sequence in A is i, where j is a function type identifier (ID). The sequence number i and set D indicate the dependency between SFC functions, in which the function with index i is dependent on the outputs of functions $i-1$ if $(f^{i-1}, f^i) \in D$. For example, in Figure 3.1, PC, FW ,and VO

Symbol	Description
L	Locations set all locations that are controlled by controller
l_k	Locations k, $l_k \in L$
p_k	l_k's CPU processing power, instruction per second
m_k	l_k's available memory, in bytes
w_k	l_k's available storage, in bytes
$d_{k,j}$	Round-trip transmission time between function f_j and location l_k
u_k	l_k's processor usage in percentage
s_j^{in}	Input data size of function f_j
s_j^{out}	Output data size of function f_j
z_k^{up}	Transmission rate for sending input data from the controller to location l_k
z_k^{down}	Transmission rate for receiving output data from location l_k at the controller

Table 3.1: Resource properties

Symbol	Description
A	Application, consist of SFC which is (F, D)
F	functions in A which is $\{f_1^1, f_2^2, ..., f_j^i\}$
D	Dependencies between A's functions, i.e., $\{(f_1^1, f_1^1)..., (f_{j-1}^{i-1}, f_j^i)\}$
n	is the number of functions in set A
f_j^i	Service function of i-th in execution and has type j
$q_{j,p}$	The number of instruction needed for f_j, integer value
$q_{j,m}$	The memory needed for f_j, in bytes
$q_{j,w}$	The storage needed for f_j, in bytes

Table 3.2: Application properties

represent functions that have an ordering in their execution: PC and FW need to execute before VO. The SFC therefore can be specified as:$(\{f_1^1, f_2^1, f_3^2\}, \{(f_1^1, f_3^2), (f_2^1, f_3^2)\})$, and f_3^2 is dependent on the output of f_1^1 and f_2^1. Table 3.1 and Table 3.2 summarises the associated resource and application properties.

Every function has execution requirements, $q_{j,p}$, $q_{j,m}$, and $q_{j,w}$, which represent process, memory, and storage, respectively, needed to execute function f_j^i. Set L represents all locations that are registered with the controller, and $x_{j,k}^i$ is an auxiliary variable that indicates execution of f_j^i on l_k, where $l_k \in L$ (i.e., f_j^i placed in l_k). p_k, m_k, w_k, and b_k are the available resources at location l_k (i.e., the process handling capacity, memory, storage, and bandwidth respectively). Processor usage and delay (i.e., between the controller and l_k), are represented as u_k and d_k, respectively. Table 3.3 summarises decision outcomes and mapping result variables.

Symbol	Description
$x_{j,k}$	Auxiliary variable indicate that f_j^i is executed in l_k value is 0 or 1
$y_{j,k}$	Auxiliary variable indicate that $T_{j,k}$ is part of the longest path value is 0 or 1
o_k	Auxiliary variable indicate that l_k is obtained by A value is 0 or 1

Table 3.3: Decision outcomes

Symbol	Description
$T_{j,k}$	Time to send and process f_j in l_k
E	The set of all placed paths of A
$Risk_{j,k}$	Risk of executing f_j in location l_k
$MaxReplicas_{i,j}$	The maximum possible replicas for f_j^i is integer
m	Constant adjust maximum possible replica is m + 1 is intege
r_k	Loss probability, the number of failures per allocations
M	The path with longest time elapses in A, $M = \{T_{1,1}, T_{2,1}, \dots T_{i,j}\}, M \in E$

Table 3.4: Decision support variable and functions

3.4.1 Estimating Completion Time

The system minimises the end-to-end latency by attempting to reduce the delay between controller fog node and deployment locations. Moreover, it estimates the time to process every function at a locations as follows:

The *network delay* between location l_k and the controller, denoted as $d_{k,j}$, represents the total time required for transmitting and receiving data for a function of type j between these two points. This delay comprises the sum of transmission times for both input and output data. Specifically, $d_{k,j}$ is calculated as $d_{k,j} = \frac{s_j^{\text{in}}}{z_k^{\text{up}}} + \frac{s_j^{\text{out}}}{z_k^{\text{down}}}$, where s_j^{in} and s_j^{out} are the input and output data sizes of the function, respectively, and z_k^{up} and z_k^{down} represent the up and down transmission rates. These rates reflect the speed of data transmission between the controller and location l_k, encapsulating factors such as the data sizes and inherent network speed.

Process node time is the time that the process node takes to complete the execution of a function in that node. This information can be obtained by benchmark the application on each process node type, which can be either SBC or VM (i.e., f_j^i in l_k). In contrast, if we acquire the function's process instruction, we can accurately estimate the processing time, presuming we have accurate data on the relevant node's processor speed. Thus, processing time of f_j^i in l_k depends on f_j^i processing requirement ($q_{j,p}$)

and l_k's processing speed (p_k) (i.e., $\frac{q_{j,p}}{p_k}$). However, it is crucial to consider that several studies [65, 66, 67, 68] have demonstrated that sustained 100% CPU utilisation can adversely affect the performance of tasks on the processing node. In this manner, it must be considered as a constraint (Equation 3.12).

Processing time of f_j^i in l_k depends on f_j^i processing requirement ($q_{j,p}$) and l_k's processing speed (p_k). The total time to execute f_j^i in l_k, ($T_{j,k}$), including transmission and processing as defined:

$$T_{j,k} = d_{k,j} + \frac{q_{j,p}}{p_k} \tag{3.1}$$

E is the set of all paths in an SFC. A path time in an SFC placement is the sum of all execution time in the path, $\sum T_{j,k}$ where $T_{j,k} \in e$. e is the sequence of execution times for services in SFC.

The longest path (M) is measured based on the longest time from the placement of the first service function to the end of the last service function within all SFC paths. The longest path is equivalent to the *makespan* of the SFC and should satisfy constraints in formula 3.9.

3.4.2 Application redundancy and cost

The controller avoids allocating a function to a location that has a high risk of failure. The risk of placing a function at a specific location is specified as: *Risk of allocating* f_j^i in l_k, i.e., $Risk_{j,k}$, is failure/loss probability times the impact of failure ($T_{j,k}$). Loss impact is $T_{j,k}$, as it is the time of the first f_j^i's allocation that failed to complete on time $T_{j,k}$, results in reallocation. Loss probability, i.e., r_k, is the number of failures per allocations, and is derived from historical l_k failure data, hence $Risk_{j,k} = r_k \times T_{j,k}$. Even after calculating the risk, there are chances of reallocating a failed function. Thus,

the system deploys replicas of the function to avoid losing time in case of failure.

$$Risk_{j,k} = r_k \times T_{j,k} \tag{3.2}$$

The redundancy of application uses a "funnel-shape" of replicated functions, as illustrated in Figure 3.4. The initially executed functions (i.e., at an early stage in the SFC) have the maximum replicas, $MaxReplicas_{i,j}$. This value decreases as we progress through an application composed of n functions, as illustrated conceptually in Figure 3.5. The constant m adjusts $MaxReplicas_{i,j}$, and $MaxReplicas_{i,j}$ does not exceed $m + 1$. Formula 3.3 is used to calculate $MaxReplicas_{i,j}$. The replication strategy is based on the observation that functions that occur at an early stage of the SFC should have higher priority (in terms of resilience), as inability to complete these successfully will cause failure downstream in the SFC. Consequently the number of replicas follow the funnel shape illustrated in Figure 3.5, where the actual number of replicas are based on the observed failure rate.

$$MaxReplicas_{j,i} = 1 + \lceil (1 - \frac{i}{n+1})m \rceil \tag{3.3}$$

The *Cost* of deploying application A is controlled by tracking the locations obtained by A in variable o_k, Table 3.4 summarises the decision support variables.

3.4.3 Problem Formulation

The main goal of this paper is to provide SFC placement aiming to minimise overall application time, deployment cost, and risk of application failure (to maximise availability). We formulate the optimisation problem as follows:

$$MIN \;\; C \;\; and \;\; MIN \;\; R \;\; and \;\; MIN \;\; O \tag{3.4}$$

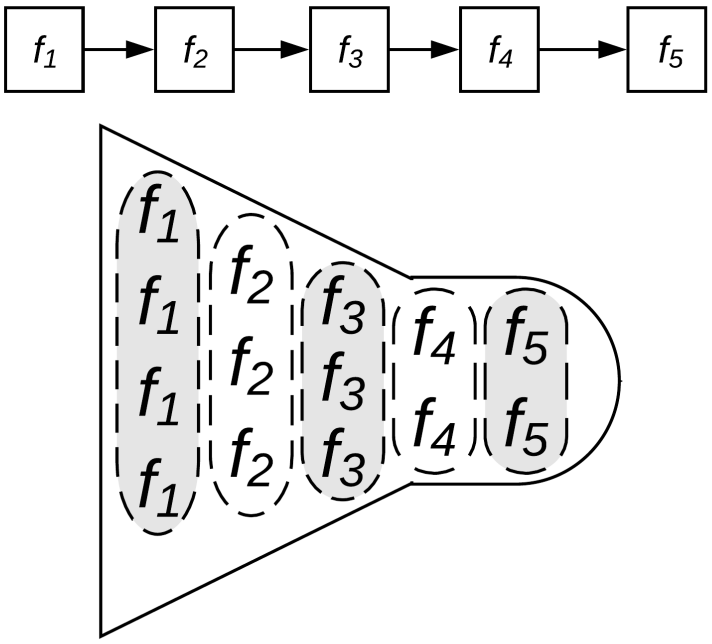


Figure 3.4: The redundancy of functions that run early in the graph has the most replicas. As the executions move through the service function chain, the number of replicas goes down. .

$$C = \sum_{j \in F} \sum_{k \in L} T_{j,k} \cdot y_{j,k} \cdot x_{j,k} \tag{3.5}$$

$$R = \sum_{j \in F} \sum_{k \in L} Risk_{j,k} \cdot x_{j,k} \tag{3.6}$$

$$O = \sum_{k \in L} o_k \tag{3.7}$$

Where objective function C (formula 3.5) is the total completion time, and objective function R (formula 3.6) is the total risk of application A completing successfully.

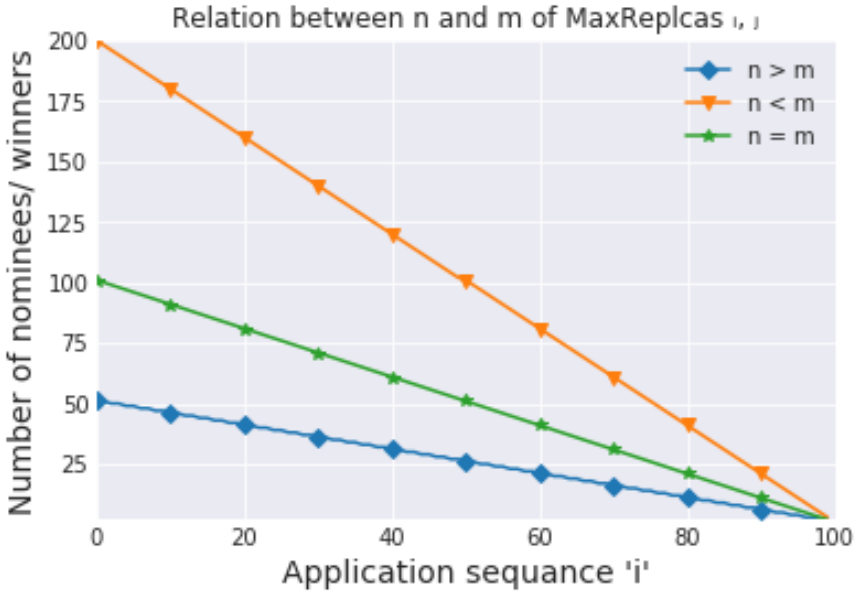


Figure 3.5: Relation between the length of the service functions chaining n and the variable m of $MaxReplicas_{i,j}$'s formula regardless of whether n or m is greater, the replicas of function i always decrease through graph execution. .

The number of locations used to execute A, including redundancy, is represented by objective function O (formula 3.7)

Subject to

$$MaxReplica_{i,j} \geq \sum_{k \in L} x^i_{j,k}, j \in F, 1 \leq i \leq n \tag{3.8}$$

$$\sum_{T_{j,k} \in M} T_{j,k} \cdot y_{j,k} \cdot x_{j,k} \geq \sum_{T_{j,k} \in P} T_{j,k} \cdot x_{j,k}, M \in E, P \in E \tag{3.9}$$

$$w_k - q_{j,r} \geq 0 \tag{3.10}$$

$$m_k - q_{j,m} \geq 0 \tag{3.11}$$

$$u_k \cdot x^i_{j,k} < 1.0 \tag{3.12}$$

$$n \geq 1 \tag{3.13}$$

$$m < |L| \tag{3.14}$$

3.5 Conclusion

This chapter proposes a method to reduce end-to-end latency by computing processing time at each location, accounting for network delay and processing node time. The problem was formulated as an ILP model that incorporates redundant deployment and prioritises the minimisation of delay, risk, and cost.

A risk assessment formula and a cost reduction method were implemented to enhance the resilience of the management process. The allocation function prioritises early-stage service functions while avoiding high-risk locations and utilising replicas to mitigate time loss in the event of failures.

Our objective function minimises resource usage across the entire SFC graph, and constraints set the maximum allowed replicas for individual functions and tasks. This results in an automated component within the system that manages replicas.

Application deployment costs depend on the resources utilised during the application's execution. This chapter established the foundation for comprehending the placement problem and the ILP techniques employed for resource optimisation. *Chapter 5* and *Chapter 6* provide additional information on the related algorithms and techniques.

Chapter 4

Toward a Platform that Supports Continuous Adaption

4.1 Introduction

The current chapter addresses the shortcomings identified in *Chapter 2*, which does not sufficiently explore the interplay of problem modelling and engineering aspects in adaptive platform research. It further develops the discussion on the placement problem from *Chapter 3*. Also, it examines the essential platform components and *engineering architecture* that facilitate efficient management of applications and infrastructure. It emphasises the role of *simulation tools* in evaluating scheduling strategies, enhancing the optimisation of platform management.

An adaptive platform for IoT applications must incorporate four key components. First, it needs an optimised scheduling planer for application execution, aided by system algorithms or optimisation tools. Second, it should manage placement actions and dataflow via a distributed processing platform, providing workflow feedback for decision analysis. Third, it requires efficient monitoring mechanisms to gather, and then cleanse, system data, including resources like RAM, CPU, and service function state. Finally, the platform should feature an analysis phase to generate insights from the monitored data, employing the ILP formula for scheduling optimisation, with the analysis output guiding the optimisation strategy.

The rest of this chapter continues as follows: Section 4.2 explains the environment and the tools that enable planning and parallel programming. Section 4.3 addresses the software and data design, focusing on how the programme is structured, how data are managed, and the adaptive processing workflow. In Section 4.4, simulation approaches for testing the scheduling and placement algorithm are discussed. These procedures represent a distributed ecosystem.

4.2 System Overview

An adaptive system is essential in the dynamic world of IoT. Advanced tools and strategies facilitate adaptability, enabling effective scheduling, placement, monitoring, and data analysis. This shapes a platform that is responsive to the continuously changing IoT environment.

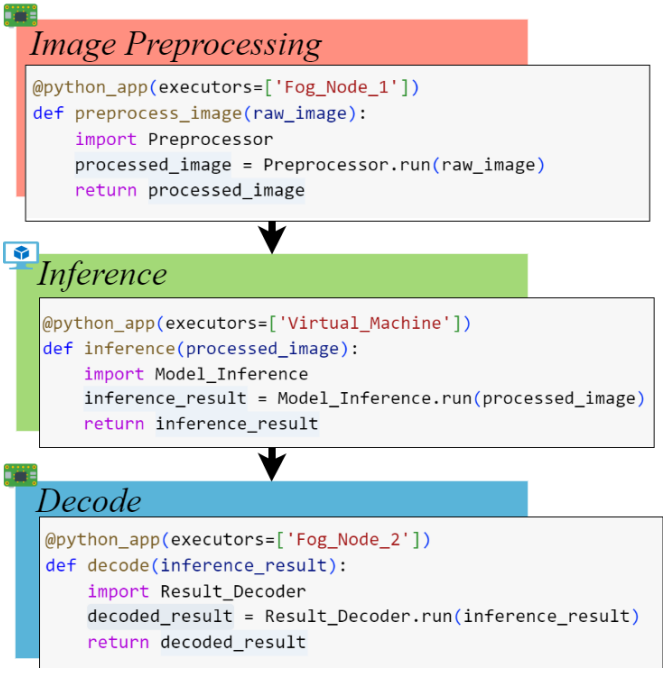


Figure 4.1: Service function chaining from the scenario shown in Figure 3.1 is implemented with Parsl. The *executors* argument in the function decorator specifies the location that runs the service function. .

Tools such as Parsl play an important part in enabling system adaptability through data streaming processing. Parsl performs SFCs in the infrastructure and produces informative logs. Parsl's use of high-level programming languages for scripting SFC, as depicted in Figure 4.1, enables an accurate representation of the SFC graph and function logic. The feedback mechanism facilitates infrastructure knowledge building (see Figure 4.2).

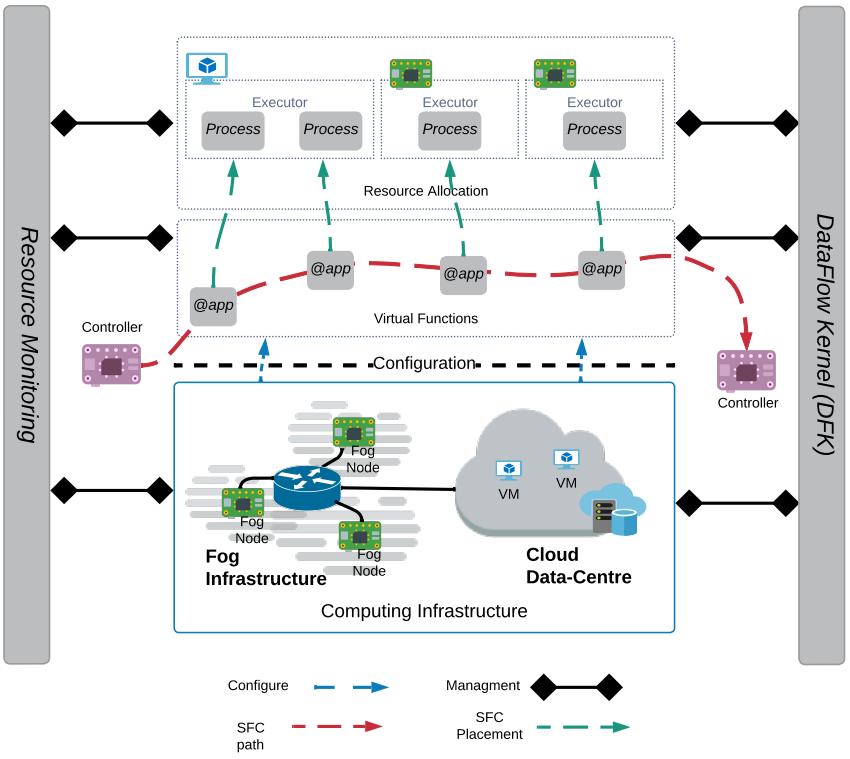


Figure 4.2: Pipelining services in a fog-cloud environment with Parsl, the controller manages the execution of the graph according to the setup configuration of Parsl's DataFlow Kernal. Processes execute the service in the process node, whereas Parsl apps store the service function's programme logic .

Parsl [9] is a Python library for parallel programming, which uses Python functions, termed *Parsl apps*, as service functions in SFC. Managed by *executors*, these apps provide asynchronous, non-blocking functions and return *AppFutures* for optimised dataflow, encapsulating the service function logic within the Parsl ecosystem.

Efficient scheduling requires evaluation of infrastructure information and past schedul-

ing results. Heuristics, meta-heuristics, optimisation solvers, and libraries such as jMetalPy [69] can be combined with the ILP model to optimise scheduling. Optimisation solvers and meta-heuristics offer different methods for addressing complex optimisation problems. Optimisation solvers utilise deterministic algorithms to efficiently solve well-structured mathematical problems, including ILP, in order to identify optimal solutions. Meta-heuristics employ stochastic and iterative methods to solve problems with large search spaces or complex objective function landscapes. Scheduling optimisation involves solvers that generate optimal schedules within defined constraints, while meta-heuristics provide adaptable solutions for dynamic environments with varying requirements.

The integration of Parsl and optimisation solver, along with their feedback mechanisms, facilitates a self-adaptive system that can adapt to changing conditions in the IoT systems (as shown in Figure 4.3). The system comprises of three components: deployments for executing SFC in the infrastructure, monitoring for providing decision-making information, and decision-making for analysing and optimising the scheduling plan using the ILP model. The platform's adaptability to the IoT environment is facilitated by its holistic approach. Table 4.1 outline the adaptive platform's system structure, data handling, and utility workflow resulting from the integration of these solutions. Table 4.2 describes each adaptive component.

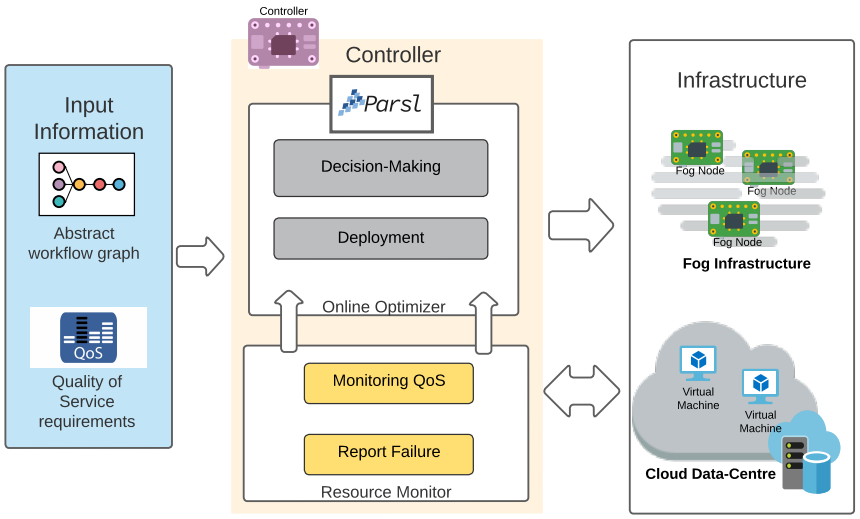


Figure 4.3: Controller supervises the placement process by collecting infrastructure data and adjusting to system situations. .

Component	Adaptive process	Tools
Monitoring	Sensors and Preservation	Stop Watch [70, 71], Psutil [72], Parsl logs [73]
Decision-making	Analysis and Planning	Data Cleaning, ILP, Optimisation Strategies (Greedy heuristics [74] and JmetalPy [69])
Deployment	Action and Actors	Parsl [73], Brokers (Apache Kafka [75], Samba [76])

Table 4.1: Adaptive component processes and tools

Component	Description
Monitoring	The monitoring component involves the use of sensors for data collection and preservation for data storage. Tools like stop watch and psutil are used for retrieving information on running processes and system utilisation, while Parsl logs capture the history of function execution and outcomes.
Decision-making	Decision-making involves analysing collected data and planning for future actions based on the analysis. Data cleaning ensures the integrity of the analysis, while the ILP model is used for utility data transformation. Optimisation strategies such as greedy heuristics and JmetalPy help in making efficient scheduling decisions.
Deployment	Deployment involves executing the actions planned during the decision-making phase. Parsl is used to execute the SFCs within the infrastructure. Brokers like Apache Kafka and Samba hold the data between the different components of the system, ensuring smooth execution of the actions.

Table 4.2: Descriptions of adaptive components

4.3 Design & Implementation

This section explains the platform's architecture and implementation in depth. In this part, data management and software design will be examined in depth. The placement algorithm and decision-making logic will not be explained in this chapter. The two chapters, Chapter 5 and Chapter 6, are devoted for the placement algorithms.

Data Type	Description
Application data	The input and output of applications that go through transformations over the course of SFC execution. The transformations can change the type of data.
Function data	Data that is generated and consumed by the application's functions. Each function accepts a specific type of data and returns a specific type of data.
Monitoring data	Data collected about the system and application state during the monitoring phase of the adaptation process.
Decision-Making data	Data used during the decision-making process, including the information about the system and application state and the formulated optimisation strategies.
Deployment data	Data generated during the deployment phase, including logs produced by data streaming tools and observations provided to the monitor.

Table 4.3: Data types

4.3.1 Distributed Systems: Data Management

The distributed system manages two main types of data: application data and utility data, and data management is essential for the system's operation. Application data includes the input and output of the system's applications. Utility data are used for monitoring, decision-making, and deployment during the adaptation process. The data is characterised by three attributes: type, locality, and passing type (Tables 4.3, 4.4, 4.5 respectively). Type refers to the specific model and structure of data that can be accepted by system components or applications. Data locality pertains to the physical location of data in memory or storage. Data passing type refers to how functions handle data transformation, either by copying the value (pass by value) or by passing the data address to the next function (pass by reference).

Data Locality	Description
In-Memory	Data is directly stored in the system's memory for quick access and manipulation. Common in application data handling and utility data generation during adaptation.
Storage	Data is stored in a more permanent storage location, such as a disk drive or distributed file system. This is common for large datasets that cannot be entirely loaded into memory.
Caching system	Data is stored in a caching system for rapid access, reducing the need to access the main storage frequently. This is useful for data that is accessed frequently.
Network streaming channel	Data is passed over the network from one function to another. This is common for large data sets or when data needs to be shared between different parts of the system.

Table 4.4: Data locality

Data Passing Type	Description
Pass-by-Value	This method involves copying the value of the data and sending it through the data streaming channel. This is commonly used when the data size is small, and the data will not be modified by the receiving function.
Pass-by-Reference	In this case, the system passes the data address to the next function, which fetches the data. This is more efficient for large data sets or when the receiving function will modify the data. Intermediary brokers such as file systems or streaming tools often facilitate this.

Table 4.5: Data passing type

Data is created and changed during the execution of SFC. The application's functions have designated input and output types, and the system transforms data as necessary.

For instance, in image processing, an image is converted into an array representing the image's pixels. Application data types are typically standard and less complex, leading to exceptions being logged if errors occur.

Utility data, generated during adaptive processes, enable system adaptations. Initially, the system collects information about the infrastructure and the QoS of the application. These data are then cleaned and prepared for capture by the decision-making process, which converts them into an optimisation strategy-accepted logical language, such as ILP modelling language. The decision-making process then produces a solution, the data of which are encoded. Encoding in heuristics can be complex and user-defined, like heaps combined with a program-defined type. Meta-heuristics typically use a standard encoding method, such as an n-array that is not specific to any particular solution. The deployment action produces two types of data logs: logs generated by data streaming tools such as Parsl, and logs that are handed over to the monitor for observation.

The system's performance is significantly influenced by how data is handled. Optimising data locality and using efficient data passing types can improve system performance by minimising unnecessary data transfer and transformation. The proper utilisation of utility data in adaptive processes enables better decision-making and more efficient deployments. Robust data handling strategies can enhance the system's flexibility and efficiency.

4.3.2 Platform Structure & Modules

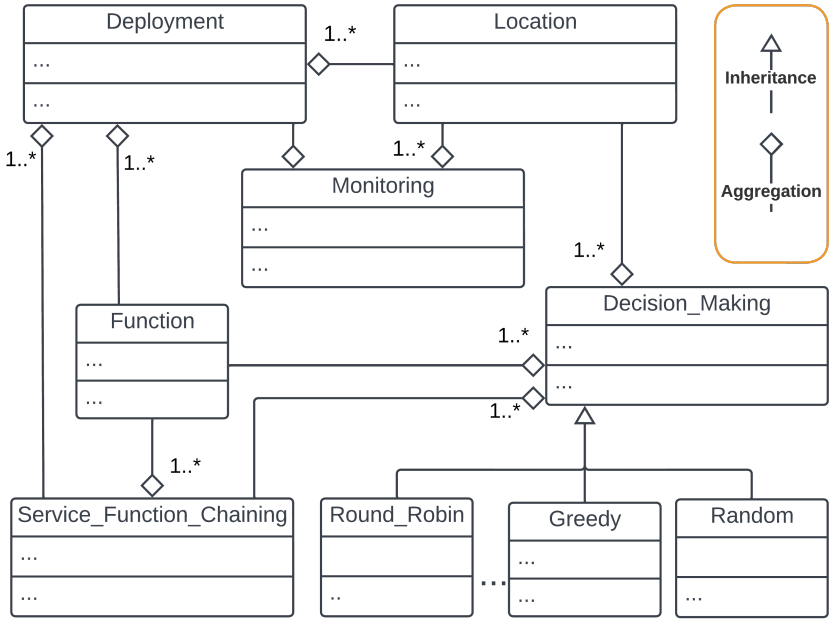


Figure 4.4: Summarised UML class diagram for the static structure of the adaptive platform .

Using Object-Oriented Analysis and Design (OOAD) principles to define the adaptive system architecture results in a modular, extendable, and maintainable design [77]. The structure, depicted in the UML class diagram (Figure 4.4), leverages encapsulation, inheritance, and polymorphism. Interactions between components like Service Function Chains (SFCs), Function, Monitoring, and Location, all depicted in Figure 4.5, ensure efficient deployment and control of SFCs.

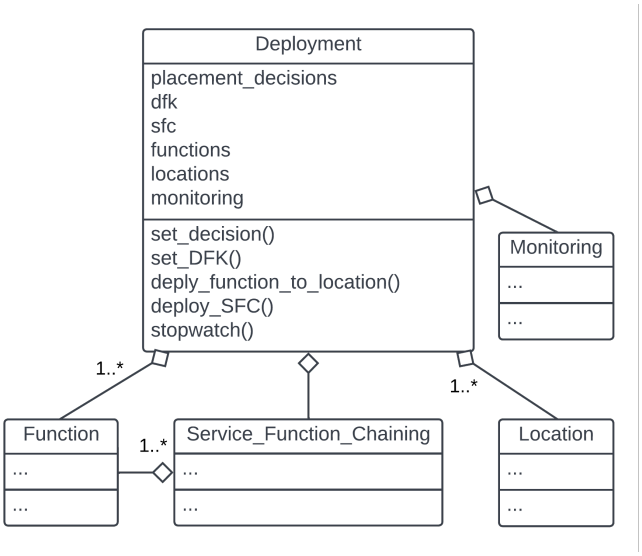


Figure 4.5: Object-oriented aggregation is used to describe a new implementation of the relationship between the *Deployment* components and other classes (such as *SFC*, *Function*, *Monitoring*, and *Location*).

At the base of the system, an abstract scheduler, detailed in Figure 4.6, lays the foundation for decision-making processes, which can include heuristic methods such as greedy algorithms. The SFC placement problem is encapsulated in the objects representing graph ordering (Figure 4.8), facilitating the evolution of the system from a basic scheduler to a complex decision-making entity.

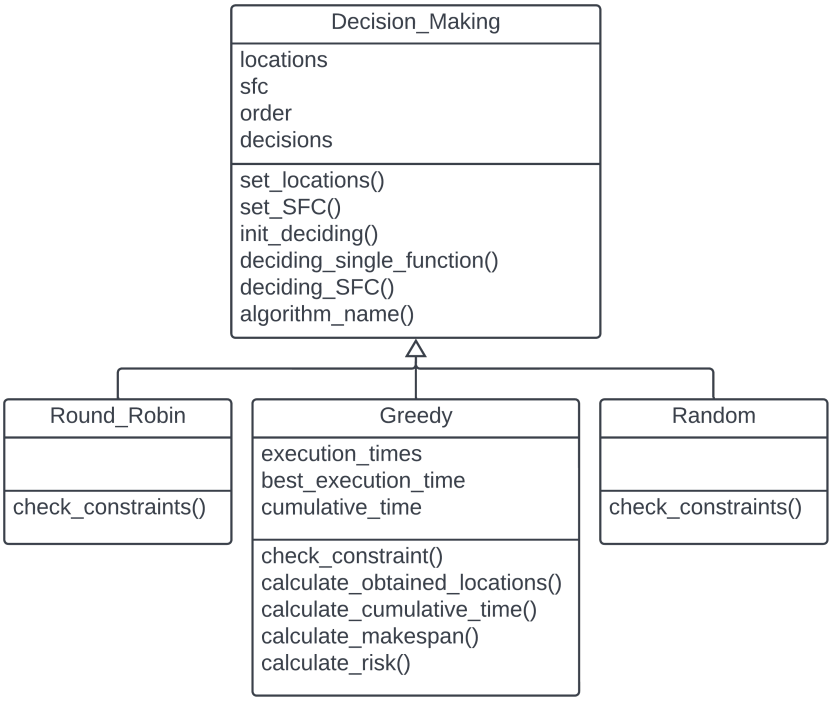


Figure 4.6: Object-oriented inheritance describes a new implementation while preserving the same behavior, allowing reuse *Decision-Making* **logic and extend it independently. .**

Interaction among system components, including monitoring, deployment, and decision-making modules, is enabled through method invocation between classes, as elaborated in Section 4.3.3. The relationship between the Monitoring component and Location (Figure 4.7), shows how monitoring facilitates performance tracking, further enhancing system responsiveness.

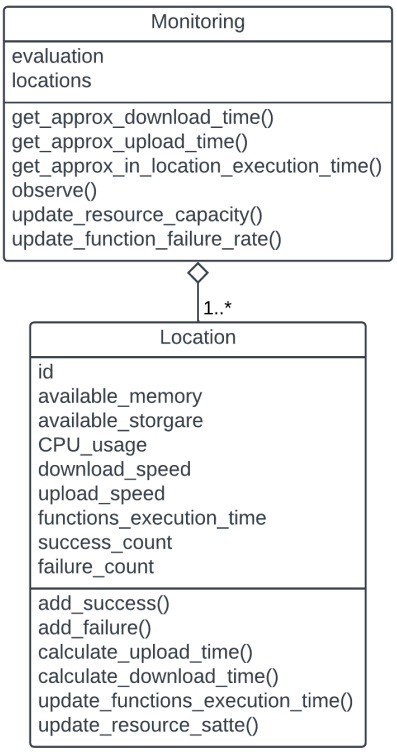


Figure 4.7: Object-oriented aggregation is used to describe a new implementation of the relationship between the *Monitoring* component and *Location*.

The Deployment component maintains a placement decision map linking functions with their execution locations (Figure 4.5). In coordination with Deployment, the Monitoring component oversees the SFC deployment process, providing real-time insights into individual function operations and overall performance.

Integration with various software tools is made possible due to the system's adaptability, with precise monitoring, improved decision-making, and efficient deployment tools enumerated in Table 4.1. The architecture's details are discussed in the preceding sections, while future sections delve into the roles and interactions of individual components.

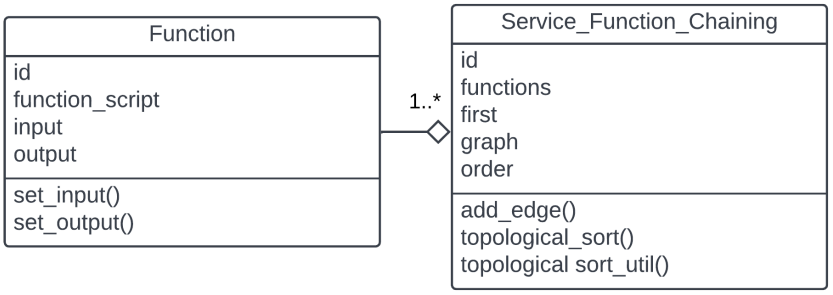


Figure 4.8: Object-oriented aggregation is used to describe a new implementation of the relationship between the *Functions* and the *SFC* class.

4.3.3 Workflow Sequence

The operation of adaptive systems is dependent on systematic collaboration. Utilising OOAD principles optimises system performance primarily through the use of static and dynamic strategies. Dynamic optimisation allows for adaptability during execution, whereas static optimisation decouples decision-making and deployment. System management determines strategy selection based on specific requirements and conditions.

Static optimisation involves deploying the SFC according to an initial schedule. This strategy uses atomic operations to organise the deployment of SFC across system components such as decision-making, deployment, and monitoring. Separate from deployment, the component responsible for decision-making processes SFC placement requests and sends a deployment plan. During execution, functions follow a predetermined order. Despite its procedural integrity, the separation between planning and execution in static optimisation limits its adaptability to dynamic environmental changes. The original plan is adhered to despite environmental changes. During deployment, the monitoring component closely monitors the duration of each function's execution to ensure that the SFC deployment occurs on time. This is shown in Figure 4.9.

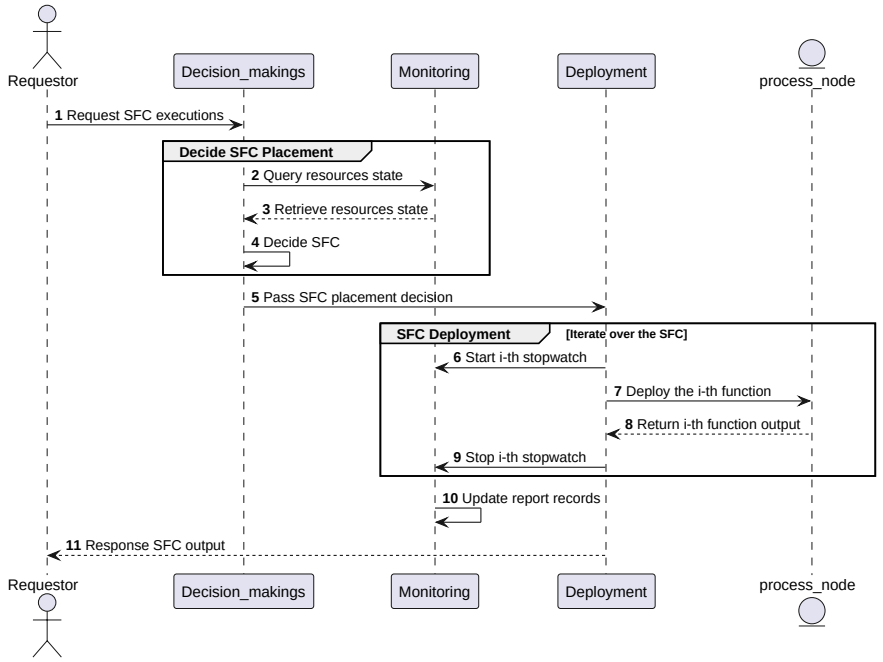


Figure 4.9: Sequential diagrams illustrate the operation of static optimisation. The deployment component does not begin deploying the SFC graph until all services within the graph have been scheduled. .

Dynamic optimisation is a technique utilised by an online optimizer within an adaptive system to adjust the SFC deployment process in response to real-time environmental changes. Unlike static optimisation, dynamic optimisation formulates the placement plan for each function individually, taking into account the current state of the infrastructure. During the deployment of each function, the deployment and monitoring components communicate frequently. Upon receiving the execution output, the monitoring component ceases operation after measuring the deployment time of each function. As depicted in Figure 4.10, the deployment status and infrastructure state are automatically updated following the deployment of each function. This enables automatic adaptation to any infrastructure changes. Dynamic optimisation increases the system's adaptability by enhancing its flexibility and responsiveness.

In Figure 4.10, a sequence diagram illustrates the operation of dynamic optimisation in the context of deploying an SFC. The diagram is divided into three main columns, each representing a different component of the process: decision-making, monitoring, and

deployment. These are the system components. Each step in the sequence is numbered and connected by arrows that indicate the flow of actions and decisions. In the diagram, a box signifies an iterative process, specifically during 'Iteration over the SFC.' When remaining functions exist within the SFC, this process requires returning to the second step in the sequence to execute these functions.

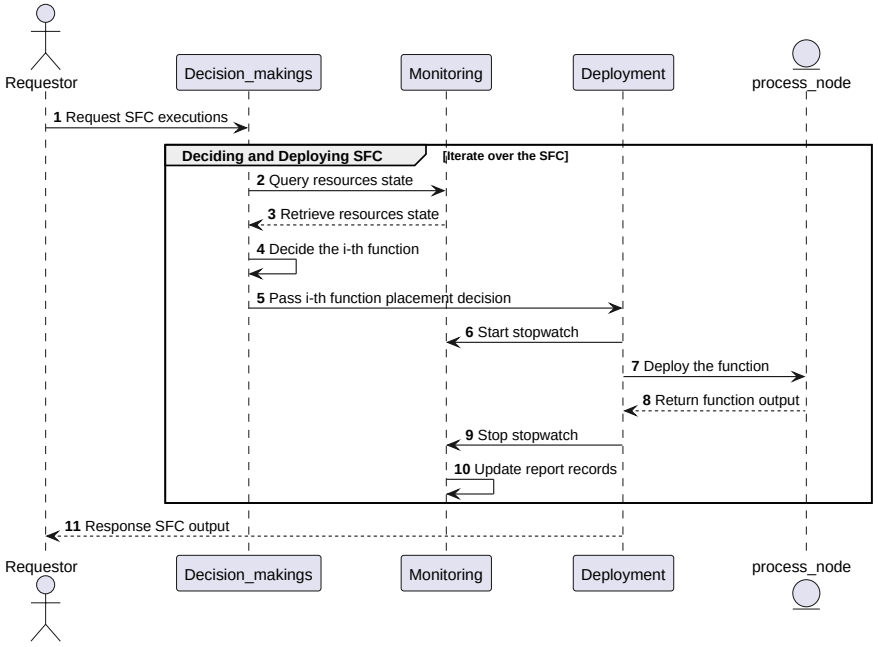


Figure 4.10: Sequential diagrams illustrate the operation of dynamic optimisation. The deployment component frequently interacts with the monitoring and decision-making components to adapt to environmental changes while deploying the SFC graph. .

The steps, starting by The requestor initiates the process by requesting SFC execution for the decision-making component (step 1). The decision-making component queries the current state of resources (step 2). It then retrieves the state of the resources (step 3). Based on the retrieved information, the decision-making component decides the i-th function (step 4). The decision on the i-th function placement is then passed on (step 5). Simultaneously, a stopwatch is started to monitor the deployment time (step 6). The deployment component then deploys the function (step 7). The function output is returned to the decision-making component (step 8). The stopwatch is stopped (step

9). The monitoring component updates the report records with the new data (step 10). Finally, the decision-making component responds with the SFC output to the requestor (step 11).

In graph theory, *topological sorting* can only be applied to DAGs. It refers to the arrangement of nodes (i.e., vertices) in a graph in which each arc moves sequentially from a preceding node to a succeeding node. Each node is only preceded by its dependent nodes or by nodes that have a directed path leading to the given node. This method is required for executing the scheduling plan and determining the SFC workflow operation sequence, as depicted in Figure 4.11.

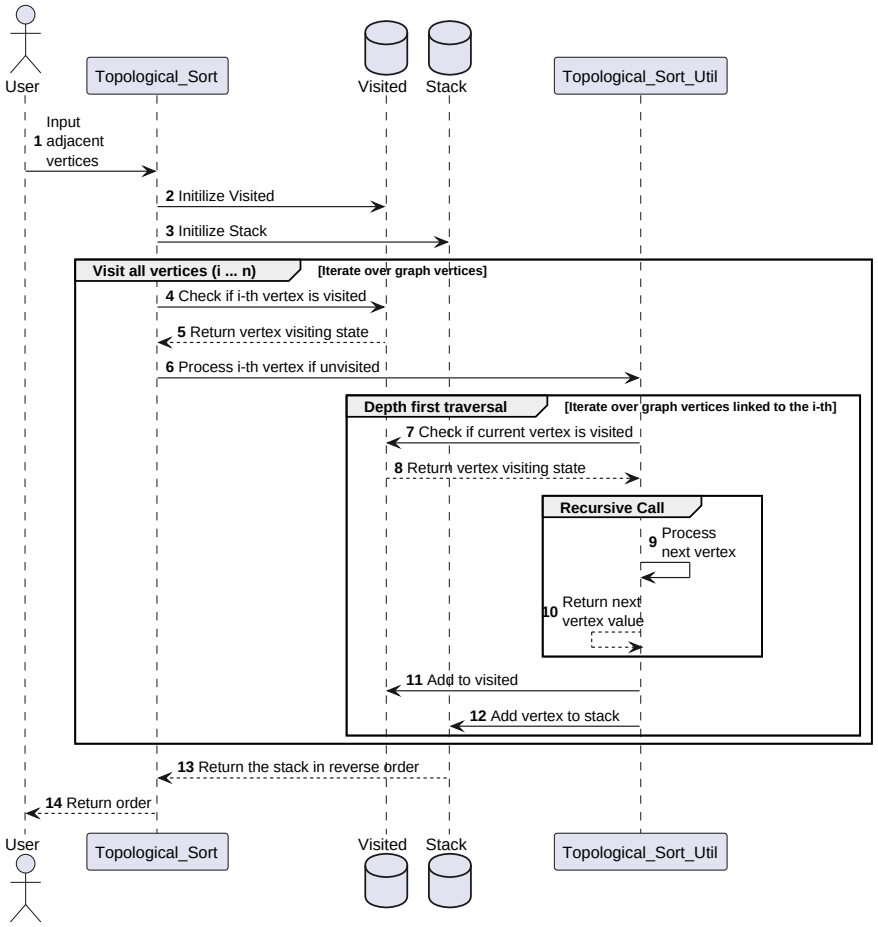


Figure 4.11: Topological sort that traverses the SFC graph. Using depth first approach, each graph node is visited only after all of its dependent nodes have been visited .

Utilising static and dynamic optimisation strategies, as well as topological sorting, the adaptive system is able to deploy SFCs effectively and respond to environmental changes. To improve the system's performance based on environmental conditions and requirements, the management of the system can choose between static and dynamic optimisation setup.

4.4 Simulation

A simulation is a computer-generated representation of the behaviour of a system over time. Models are required for simulations; the model depicts the essential traits or behaviours of the chosen system or process, while the simulation depicts the model's evolution over time. Computers are frequently used to run simulations [78]. To develop discrete-event simulations, it is necessary to represent the arrival of a function request and its subsequent departure from a process node. Request-Arrival and Output-Departure are two of the system events. The logic of arrival and departure events includes the beginning of the function's execution in the process node. The system states that are affected by these events include the status of the process node, such as the resources available and whether or not the node has capacity. In the following, we will describe some of the simulations that imitate environmental behaviour.

4.4.1 Profiling Systems with Synthetic Data

Synthetic data, generated through computer simulations, serves as a vital tool for evaluating system performance. It models system conditions, providing a practical platform for testing and fine-tuning decision-making algorithms across a variety of system conditions and IoT application characteristics. This mimics the behaviour of actual applications and infrastructure drives system decisions, which provides accurate distributed system behaviour profiles (Figure 4.12).

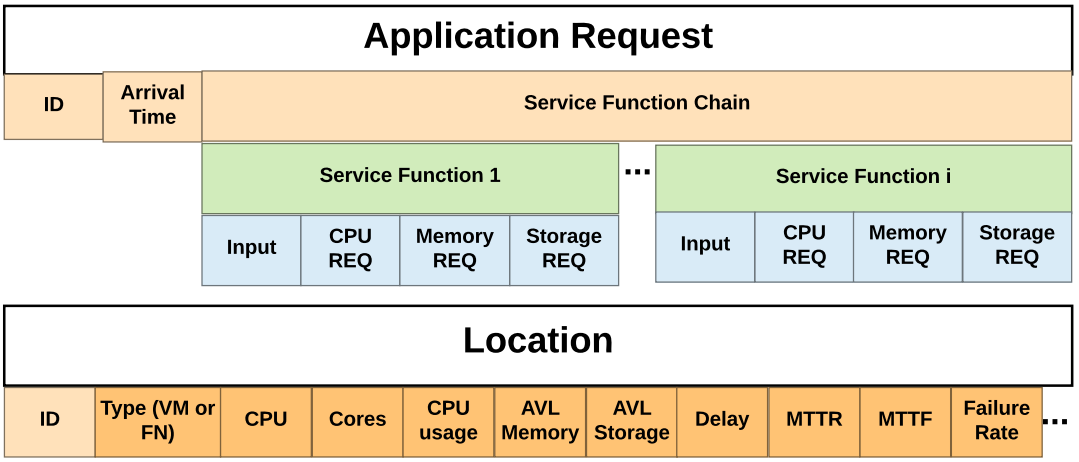


Figure 4.12: Synthetic SFC requests are generated with this format during the simulation. .

Simulated *location characteristics* (i.e., process node characteristics) include dynamic computational attributes such as capability, capacity, delay, and available resources, which are frequently updated to mirror system behavior. This synthetic data is crucial for understanding each process node's performance under different conditions.

Function requirements are simulated by generating unique *application requests*, each associated with an arrival time and service functions. These requests mimic data streams directed to a single controller in the fog infrastructure, encapsulating the demands on the process node.

4.4.2 Clock-Based Failure Model

The failure model focuses on processing nodes or locations where the simulation employs clock mechanisms to generate failure. It is mainly employed to assess platform decisions based on the occurrence of location failures. The model does not account for software errors, which may result from imperfect service function implementations. We adopted a similar approach to those papers that studied failure and modelled it, with a specific focus on the timing of when failures occur [79, 80]. Table 4.6 provides

a comprehensive summary of the failure model employed in this work.

Names	Description
Failure	A period when a location becomes unresponsive.
Recovery	Restoring after freezing, i.e, failure
MTTF	The time after recovery to location's failure
MTTR	The restoration period of a location post-failure.
MTBF	Period of time between two failures
MTBF-clock	A location's cycle, split into MTTF and MTTR.
Execution time	The time to run function in an location
Waiting time	The time where functions waits the location to recover

Table 4.6: Failure mode definitions

A location's failure is defined as the period during which it fails to respond to function execution. This inactive state causes a further delay until the node recovers, which is known as the recovery time. Mean Time To Failure (*MTTF*), which represents the average time until the next failure, and Mean Time To Recovery (*MTTR*), which represents the average time from failure to *recovery*, are integral parameters associated with this concept.

Mean Time Between Failures (*MTBF*) is the average time of *MTTF* and *MTTR* events, *MTBF* is computed as the sum of *MTTF* and *MTTR*. Every location has an *MTBF-clock* that measures the local MTBF time. This clock resets to zero after each failure and begins tracking the subsequent *MTBF* interval from beginning. It begins by measuring the *MTTF* period, then the *MTTR* period.

While executing a service function within the *MTTF* period and assuming there are no interruptions (i.e., the request does not overlap with the *MTTR* period), the location is capable of processing the function within the anticipated execution time. However, if a function occurs during the *MTTR* period, execution is delayed until the end of

the *MTTR* period. Figure 4.13 illustrates service functions submitted across various *MTBF-clock* periods.

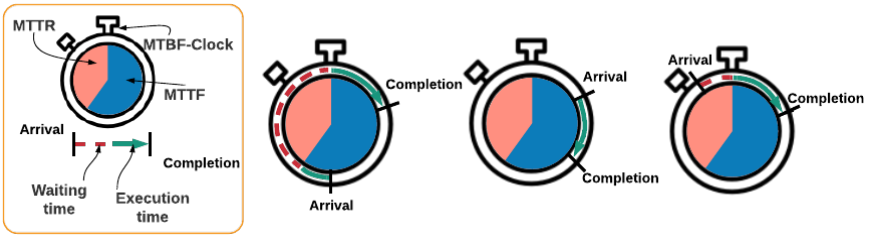


Figure 4.13: The same virtual function's execution at a location shows a completion time for three different arrival times .

Algorithm 1 (line 2 and 3) calculate the *MTBF* and the remaining period to the next cycle, MTBFREMAIN. ARRTIME is the timestamp at which a request arrives. If a request arrived within *MTTF*, then the method checks *MTTF* has sufficient time to execute the requested function, EXECTIME, otherwise it is considered a failure (if statement in line 4). The method determines the COMPLETIONTIME of the function (line 6-12). Output of the FINISHINGTIME method determines allocation state, whether a failure has occurred and the actual finishing time of service functions.

Algorithm 1 Evaluating completion time by integrating a failure model

```
1:  method FINISHINGTIME ( MTTF, MTTR, EXECTIME, ARRTIME )
2:      MTBF = MTTF + MTTR
3:      MTBFREMAIN = ARRTIME % MTBF
4:      if MTBFREMAIN < MTTF AND |MTTF-MTBFREMAIN| ≥ EXECTIME then
5:          ALLOCSTAT = TRUE
6:      if ALLCSTAT then
7:          COMPLETIONTIME = EXECTIME + ARRTIME
8:      else
9:          if MTTF ≥ EXECTIME then
10:             COMPLETIONTIME = ARRTIME + EXECTIME + |MTTF-MTBFREMAIN|
11:         else
12:             COMPLETIONTIME = ∞
13:     return ALLCSTAT, COMPLETIONTIME
```

4.4.3 Node Performance Degradation

In this simulation, the time-dependent reliability of a process node in an IoT application is analysed using a model based on a time-dependent failure probability framework. The *Weibull distribution* [81] is the foundation of this simulation, providing an adjustable way to evaluate and predict the performance of a node and its potential failure points over time. Certain aspects, such as delays in service functions caused by node failures, are factored into the simulation. Table 4.7 lists the simulation's Weibull distribution parameters.

Parameter	Description
Start time	Time when a node starts a function
Current time (x)	Elapsed time since the *Start time*
Time-to-failure (λ)	Duration until a node's services fail
Reliability variable (k)	Weibull shape parameter indicating reliability

Table 4.7: Weibull distribution parameters

The *start time* is an essential parameter in this simulation, as it specifies when a node's function begins. This commencement could occur either at the outset of a recovery phase or after its conclusion. The closer a node is to this *Start time*, the more efficient it is. As the node moves away from the *start time*, its service gradually degrades.

Current time (x) measures the duration of the system's operation. This is the elapsed time since the *start time*, providing a dynamic metric for determining the system's operational duration.

The simulation includes a *time-to-failure* parameter (λ) that represents the expected lifetime before a node's services are non-operational. The λ parameter characterises the scale of the Weibull distribution and is defining the life span or *scale* of the process node. Its estimation models service disruptions overtime. The higher the value of λ, the longer we can expect the node to operate without failure.

The degree of dependability during *current time* is quantified by the reliability variable (k), which represents the Weibull *shape* parameter.This variable is a measure of the node's dependability over time. A greater k value indicates greater dependability. The Weibull distribution provides a robust method for evaluating and predicting the node's performance and potential failure points over time.

The parameters, such as starting time x, scale λ, and shape k, are utilised to calculate the probability of failing to complete within the deadline. The probability of failure, denoted by $f(x; \lambda, k)$ (utilising Formula 4.1), determines whether a function meets

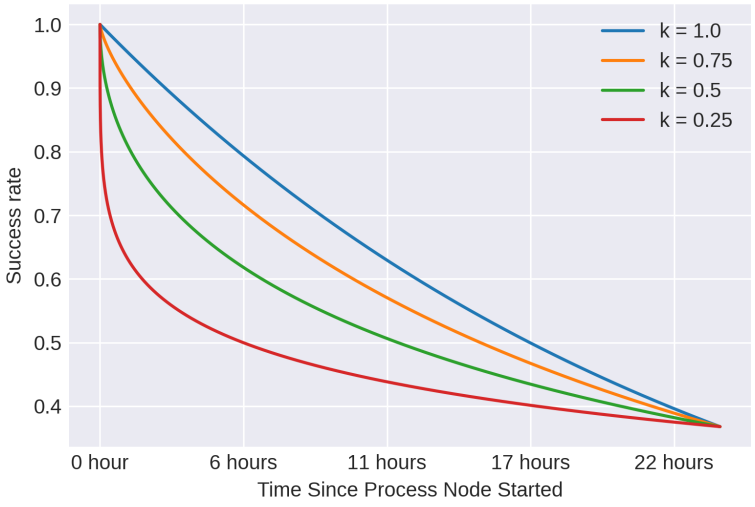


Figure 4.14: Four different process nodes, i.e., locations, performance degrades over time. Here, the scale parameter, λ, is 24 hours according to the Weibull distribution. .

its deadline or not, via a random choice mechanism. Figure 4.14 shows the different probabilities of failures according to the various parameters set up.

The simulation uses a Weibull distribution-based random function that generates failures. First, the system receives task deployments and assigns them to resources. Next, a random function decides whether the task fails or succeeds. The placement algorithm then tracks these outcomes to assess the reliability of the resources. This information helps the placement algorithm in future deployments, serving as a proactive fault tolerance mechanism.

$$f(x; \lambda, k) = 1 - \mathrm{e}^{-(x/\lambda)^k} \tag{4.1}$$

Algorithm 2 is an organised method for calculating the time it takes for a function to be completed, using the Weibull failure probability principles. The algorithm starts by estimating the function's completion time (line 4), assuming a perfect scenario without any expected failures. Afterwards, probabilities related to both failure and success

Algorithm 2 Failure model based on Weibull distribution

```
1:  method FINISHINGTIME ( λ , k , STARTTIME, CURRENTTIME, EXECTIME )
2:      ARRTIME = CURRENTTIME
3:      x = STARTTIME
4:      COMPLETIONTIME = ARRTIME + EXECTIME
5:      SUCCESSRATE = 1 − e^{−(x/λ)^k}
6:      FAILURERATE = 1- SUCCESSRATE
7:      FIRSTALLCSTAT = RANDOMCHOICE(SUCCESSRATE, FAILURERATE)
8:      ALLCSTAT = FIRSTALLCSTAT
9:      if SUCCESSRATE > 0 then
10:         while ALLCSTAT IS FALSE do
11:             COMPLETIONTIME = COMPLETIONTIME + EXECTIME
12:             ALLCSTAT = RANDOMCHOICE(SUCCESSRATE, FAILURERATE)
13:     else
14:         COMPLETIONTIME = ∞
15:     return FIRSTALLCSTAT, COMPLETIONTIME
```

scenarios are calculated and revised (line 5 and 6). The algorithm uses probabilities and a random function to decide the allocation state of the process node (line 7). When the success probability reaches zero (not satisfying condition in line 9), the node is considered non-operational and will not generate any results (line 14). If the allocation state is unsuccessful, more time is added to the completion time (line 11). Then, the random function is run again to check if the state has become successful (while loop in line 10). The process iterates until a successful state is achieved (line 10 to 12). The algorithm returns the placement state and its execution time at a specific location (line 15).

4.4.4 Link Quality in Mobile Edge Device

Mobile edge devices play a role in modern communication systems, enabling data processing at the edge of networks. To optimise network performance, understanding the dynamics of their communication is essential. In this simulation that models edge device movement as a stochastic process, providing insights into link quality and latency variations. We adopted a similar approach to those projects that studied meth-

Algorithm 3 Simulation of a random walk

```
1: method RANDOMWALK ( STEPSNUMBER, CURRENTLOCATION )
2:     (x, y) = CURRENTLOCATION
3:     for 1 ... STEPSNUMBER do
4:         (dx, dy) = RANDOMCHOICE([(0, 1), (0, -1), (1, 0), (-1, 0)])
5:         x = x + dx
6:         y = y + dy
7:         CURRENTLOCATION = (x, y)
8:     return CURRENTLOCATION
```

ods on searching efficiency in swarm robots for area exploration [82]

The simulation employs a random walk to model the movement of edge devices. Each iteration projects the device's coordinates onto a 2D plane/grid. A random walk starts by selecting an arbitrary number of steps, with each step allowing movement in north, south, east, or west directions. This is reflected in Algorithm 3, line 4, where each step denotes a movement in a single direction and is represented by an increment of one to the current x or y position Algorithm 3, line 4 to 7).

After calculating the distance between the edge device and fog nodes, This distance is integral to evaluating the wireless connection quality. The connection quality is determined not only by distance but also by factors such as frequency, obstacles, and signal fading. These elements collectively affect the path loss and signal-to-noise ratio (SNR), which are critical in determining the effective capacity of the wireless link.

Parameter	**Default Values**
Frequency	2.4×10^9 Hz
Obstacle Factor	1.0
Bandwidth	10^6 Hz
Noise Power	10^{-9} Watts
Data Size	10^6 bits
Fading Coefficients	Rayleigh distribution (scale=1, size=100)

Table 4.8: Parameters and their default values for our *Wireless Simulation*

The simulation incorporates a detailed model for signal propagation and fading (values specified in Table 4.8). *Path loss* is calculated using both the free-space path loss (*FSPL*) formula (Equation 4.2) and additional loss due to obstacles, which is a function

of distance. The formula for *path loss* is given by Equation 4.3 [83].

$$\textit{FSPL} = 20\log_{10}(\textit{Distance}) + 20\log_{10}(\textit{Frequency}) - 20\log_{10}(c) \tag{4.2}$$

In wireless communication, c represents the speed of light in a vacuum, roughly 3×10^8 metres per second, essential for signal propagation calculations.

$$\textit{Loss dB} = \textit{FSPL} + \textit{Additional Loss} \tag{4.3}$$

$$\textit{Additional Loss} = (\textit{Adjusted Exponent}) \cdot (\log_{10}(\textit{Distance})) \tag{4.4}$$

The model also considers *Rayleigh fading* to simulate the impact of multipath propagation. The adjusted fading coefficients for path loss are computed as in Equation 4.5 [83].

$$\textit{Adjusted Fading Coeffs} = (\textit{Fading coefficients}) \cdot (\textit{Loss Linear}) \tag{4.5}$$

where *Loss Linear* (Equation 4.6) is defined as:

$$\textit{loss linear} = 10^{-\frac{\textit{loss dB}}{10}} \tag{4.6}$$

Using these parameters, the simulation applies *Shannon's formula*, as shown in Equation 4.7, to estimate the capacity of the wireless link [83].

$$\textit{Capacity} = \textit{Bandwidth} \cdot \log_2(1 + \textit{Average(SNR)}) \tag{4.7}$$

Taking into account the bandwidth of the channel and the average *SNR* under fading conditions. The time to send data over the network is then calculated using Equation 4.8.

$$\textit{Transmission Time} = \frac{\textit{Data size}}{\textit{Capacity}} \tag{4.8}$$

This approach provides a more realistic estimation of the transmission time in a wire-

less network, considering various environmental and technical factors. The updated simulation moves beyond a simple distance-based quality metric to a more comprehensive model that includes path loss, fading, and SNR, offering a nuanced understanding of wireless network performance in different scenarios.

4.5 Conclusion

In conclusion, this chapter offers a comprehensive examination of the software design, testing procedures, and operational strategies for a platform optimised for real-time applications in dynamic, distributed environments such as the IoT and Fog-Cloud ecosystems. The development and implementation of an adaptive control mechanism, integral to rapid adaptation to the fog-cloud environment's requirements, are detailed.

The chapter further explores the modification and reuse of a placement component in compliance with user-defined decision algorithms and defines the management of data transformations via varied data models and transmission procedures. The importance of dynamic optimisation in the platform's operation is underscored, with a focus on the use of forward dynamic programming techniques for the operation of the SFC graph– an approach well-suited to real-time, event-based systems.

The chapter also presents a set of tools for evaluating the system's decision-making capabilities. These models simulate the ecosystem's operation, providing essential knowledge for enhancing platform efficacy and creating a model that assists in the test-bed environment.

The chapter describes the engineering, development, deployment, and operation of the adaptive platform. However, the basic logic behind the decision-making process for SFC placement scheduling requires further exploration. *Chapters 5* and *6* focus on creating scheduling algorithms for SFC placement, further elaborating on this fundamental part of the platform.

Chapter 5

Greedy Nominator Heuristic (GNH): Harnessing MapReduce for Function Placement

5.1 Introduction

Chapter 4 provides an overview of the platform and how it manages the real-time applications. Also, it highlights the three foundations of controls: monitoring, decision-making, and deploying. This chapter focuses on decision-making in scheduling. Also, it extend the decision-making component established in the *Chapter 4*. A greedy approach [74] is a natural extension of the platform; since it leverages dynamic programming to traverse the service function graph (SFC)

This chapter provides a fast scheduling algorithm for the SFC placement problem. The algorithm uses parallel computing to speed up the scheduling process. The main objectives of the scheduling process are low delay, low cost, and low risk. The scheduling process analyses the trade-off between the three objectives and decides the placement plan based on the analyses.

The Method for order preference by similarity to the ideal solution is utilised to do a trade-off analysis of the placement's optimisation objective, also known as *TOPSIS*) [84]. The algorithm evaluates function placement to the optimal placement with

respect to makespan, risk, and cost. The TOPSIS method is also the heuristic function [18] of the greedy method, which directs the search for the optimal solution.

The algorithm we propose uses *MapReduce* to speed up decision variable analysis. MapReduce is a parallel programming model that divides the analytic workload on a computing cluster to speed up the analytic process [64]. The scheduling time is included in the end-to-end latency of the application. Accelerating the analytics speed the scheduling process, hence reducing the completion time.

The remaining sections of this chapter will be organised as follows: Section 5.2 outline the research methodology. Section 5.3 is about the algorithm, and the component of the algorithm will be described in-depth. Section 5.4 put the algorithm's optimisation performance to the test, comparing it with a state-of-the-art approach. Finally, section 5.5 will bring the chapter to a close.

5.2 Methodology

In this section, we describe the requirements for scheduling process, the criteria for evaluating solutions, and the evaluation approach.

5.2.1 Scheduling Requirements

The scheduling algorithm must be fast and provide a high-quality placement plan. It must be built on top of the architectural solution in *Chapter 4*. Also, it must optimise the placement based on the problem definition in *Chapter 3*. The size of the infrastructure and the hardware that executes the scheduling algorithm affects the time required to make a placement decision. Therefore, the scheduling algorithm must utilise its resources to speed up the planning process.

A high-quality placement plan requires the scheduling strategy to be aware of the time, risk, and cost of the application.

5.2.2 Evaluation Criteria

The evaluation is based on four criteria: *scheduling speed*, *makespan*, *risk of non-completion*, and *cost*. Scheduling speed is evaluated across a range of scheduling process configurations and decision variable sizes. Then, compare the scheduling with a state-of-the-art meta-heuristic approach, which is particle swarm optimisation (PSO).

5.2.3 Test Environment

The experimental framework aims to understand and evaluate the performance of the scheduling algorithm.

The test bed evaluation considers the three criteria of the scheduling algorithm: makespan, risk of non-completion, and cost; under-simulated unreliability in the infrastructure. The simulation introduces faults in the system over the course of a day. We use the tools from *Chapter 4*, which are synthetic data in section 4.4.1 and a failure model in section 4.4.2 for the evaluation. Regarding the speed of the scheduling process, we need to measure the time between the start and the end of the algorithm's execution. We use *Python time* to measure scheduling process time. Since the data location affects the performance of MapReduce, the evaluation is done in a variety of data locality set-ups, such as main memory and file systems.

5.3 GNH Algorithm

The optimisation algorithm is based on the search for optimal solutions for each function in an application. The search yielded the discovery of the "optimal deployment strategy" for the application. GNH employs MapReduce to identify potential locations for redundant deployments. Workers (i.e., Mappers) loop through the decision variables. Mappers' result are sent to the Reducer (i.e., control fog node), whose results determines the best locations for redundant deployment across the all locations.

The components that enable GNH are detailed in the subsequent section, which is section 5.3.1.

5.3.1 Algorithm Components

The **parallel** model's execution is synchronisation-based, which means that *Reducer* waits for all *Mappers* to complete tasks before starting to run. Parsl [9] supports a variety of methods for executing functions and transferring data, including shared memory and file systems. GNH can utilise both ways to handle data passing between mappers and the reducer.

A **similarity function** is used to compare the general solution to an ideal solution [85]. Usually, it is a norm function, for example, Euclidean distance is 2-norm. For all functions, l_{ideal} is the location that has zero execution time and no risk, and no additional locations obtained by application A, i.e., O_{ideal}, and is also represented as point $(0, 0, O_{ideal})$. GNH uses Euclidean Distance ($ED_{j,k}$), shown in formula 5.1 in the Mapper to compare l_k with l_{ideal}[1]. A number of other measures may also be used to perform similarity comparison, such as applying fuzzy metrics that capture a degree of membership. Our implementation can be generalised and extended to use other measures also, as the distance measure can be application- and context-dependent.

$$ED_{j,k} = \sqrt{(0 - T_{j,k})^2 + (0 - Risk_{j,k})^2 + (O_{ideal} - O_k)^2} \tag{5.1}$$

Max-heap is a complete binary tree that is used to store the Mapper(s) and Reducer results. The root of the tree has the maximum value in the tree, and the value decreases as we move to lower levels in the tree. Max-heap is of $MaxReplica_{i,j}$ size, where each node has key-value pair $\langle key; value \rangle$ (e.g., $\langle result; l_k \rangle$ in algorithm 4). The key is the $ED_{j,k}$ result, whereas the location is the value.

[1] O_{ideal} is simply the number of locations that executed the previous functions in A. Therefore, O_k in the next function is equal to O_{ideal} or O_{ideal} incremented by one. The incremented value is multiplied by a weight to have a higher impact on $ED_{j,k}$

Algorithm 4 Mapper receives L and f_j^i and Return $MapperResult$ of size $MaxReplicas_{i,j}$

```
1: class MAPPER
2:     method MAP (L; f_j^i)
3:         count ← 0
4:         for all l_k ∈ L do
5:             result ← ED(l_k, f_j^i)
6:             if count < MaxReplicas_{i,j} then
7:                 INSERT(⟨result; l_k⟩)
8:                 count ← count + 1
9:             else
10:                if GETMAXRESULT(MaxHeap) > ⟨result; l_k⟩ then
11:                    INSERT(⟨result; l_k⟩, MaxHeap)
12:                    REMOVEMAXRESULT(MaxHeap)
13:                MapperResult ← MaxHeap
14:        return MapperResult
```

Mappers apply the $ED_{j,k}$ function to all locations, and then store them in Max-heap of $MaxReplica_{i,j}$ size. Max-heap order is based on the key (i.e., $ED_{j,k}$ result), not the values (i.e., locations). Every Mapper keeps a local record of the locations they monitor, and the records are results of monitoring the computing resources and the network connections linking these locations.

The **Reducer** receives results from every Mapper, concatenates them, and applies a Max-heap INSERT then REMOVEMAXRESULT functions to each location in the Mappers' results. Finally, the Mappers' results are reduced in a single Max-heap that has the locations (in the value of $\langle key; value \rangle$ in algorithm 5) to deploy the current function, i.e., f_j^i in the requested application A.

5.3.2 Algorithm Workflow

The system has a controller that plays the role of a reducer, whereas workers in Figure 3.2 (in *Chapter 3*) are Mappers. In Figure 5.1, Mappers monitor network performance and available computing resources at specific locations. Moreover, the system nominates locations to execute service functions. Whereas, the Reducer chooses from

Algorithm 5 Reducer receives $MapperResult$ and Return MAXHEAP of size $MaxReplicas_{i,j}$

```
1:  class REDUCER
2:      method REDUCE (MapperResult)
3:          count ← 0
4:          for all ⟨key; value⟩ ∈ MapperResult do
5:              if count < MaxReplicas_{i,j} then
6:                  INSERT(⟨key; value⟩, MaxHeap)
7:                  count ← count + 1
8:              else
9:                  if GETMAXRESULT(MAXHEAP) > ⟨key; value⟩ then
10:                     INSERT(⟨result; l_k⟩, MAXHEAP)
11:                     REMOVEMAXRESULT(MaxHeap)
12:         return MaxHeap
```

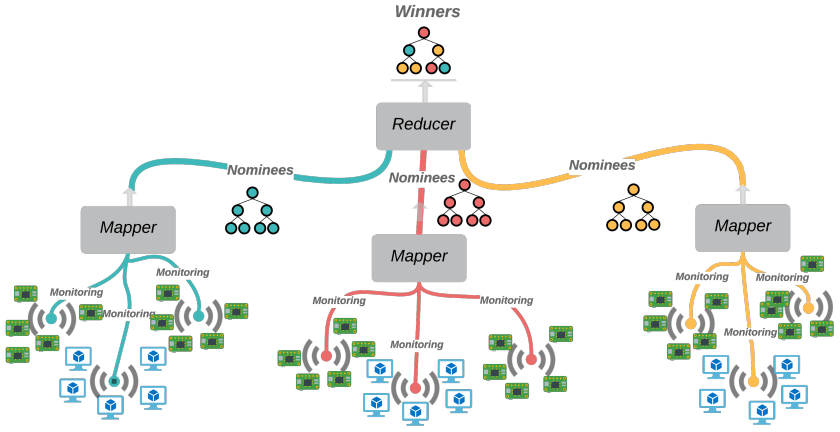


Figure 5.1: MapReduce performs GNH. Each mapper has a group of locations to monitor, and each group has its own colour (green, red, and yellow). The final Max-heap has a variety of node colours due to them coming from different mappers.

the nominated locations to place redundant function instances. This is achieved by running MapReduce using Parsl.

Both Mapper and Reducer algorithms 4 and 5 use similar search mechanisms: they loop through the search space and update Max-heap. However, the difference is that Mappers apply the $ED_{j,k}$ function (line 5 of algorithm 4), then compare the result with the worst location within the Max-heap, that is, the peak or root of the tree. Both

algorithms initialise Max-heap tree of $MaxReplicas_{i,j}$ size (lines 2-8 in algorithm 4 and lines 2-7 in algorithm 5). Inside the for loop (line 4 in algorithm 4 and 5) the new results ($result$ in mapper, and key in reducer) are compared with the peak of the Max-heap, if the result is less then the peak, then the new result replaces the Max-heap.

5.4 Evaluation

This section evaluates GNH's decision-making speed and outcomes in a controlled environment. Subsequent assessments include the algorithm's execution efficacy and the quality of its solutions in terms of makespan, risk, and time.

5.4.1 Speed Performance Evaluation

In this section, we assess GNH's decision-making speed under different control setups, beginning with the experiment setup and followed by its results.

Experiment Setup

Both the controller and workers (i.e., reducer and mappers) are Raspberry Pi 3 models B+. The experiments are carried out in various MapReduce setups with varying mapper numbers and data locality. We tested the algorithm in two infrastructure configuration one with 1,000 locations, and another with 100,000 locations. Since Parsl allows decisions to be processed in parallel, we evaluate speed with a single service function. We also considered locality of data, whether they are in a file or in memory.

Results

As can be seen in Figure 5.2, the time to make a decision is not affected by the number of replicas. This is due to the efficiency of the Max-heap operations which reduce the

time to compare locations.

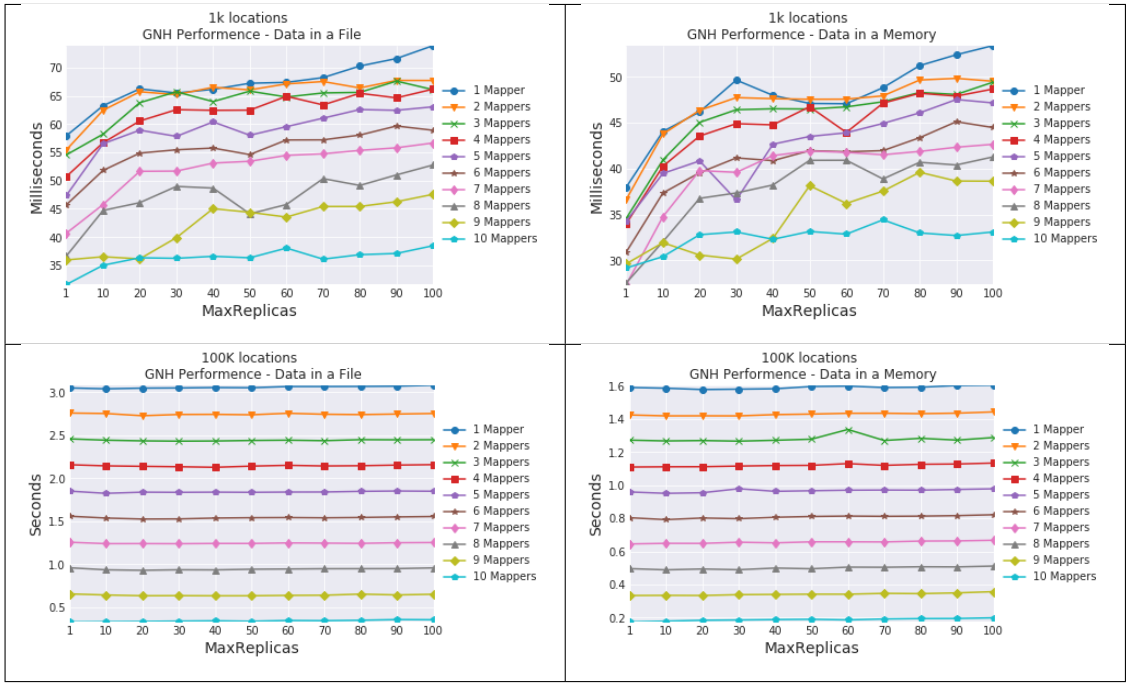


Figure 5.2: GNH performance

With less than 1,000 location, adding a new mapper to the GNH can increase the performance from 2% to 15%. Moreover, dividing the decision variables into 10 disjoint sets, in which each mapper has 100 locations, can speed up decisions to about 48% when data are stored in a file. However, if the data are already in memory it is faster by about 35%.

When 100,000 locations are considered the decision-making overhead has a more significant impact. Every added mapper can boost the performance from 10% to 45 %. Moreover, 10 mappers in the GNH is faster than single mapper by 88%.

Every time we add a mapper the performance improves by 10-14%. Until we add the 5th mapper the performance exceeds 15%. Addining additional mappers continues to improve performance. For example, 6 mappers outperform 5 mappers by 20%, 7 mappers outperform 6 mappers by around 25%, and 10 mappers outperform 9 mappers by 47%.

Loading a file with 1,000 locations to memory takes between 8ms (milliseconds) and 23ms . Therefore, keeping the data in memory can boost the decision-making speed by up to 40%. However, loading data from files to mappers will not take much time if there are fewer than 100 locations. On the other hand, with 100,000 locations a single mapper can take 1.25 seconds to load the locations to memory, dividing them between 10 Mappers reduces it to 100ms.

Files with less than 100 locations are easily loaded in memory with minimal delay. Therefore, it would be better to divide the locations recorded into files that can be loaded to the memory concurrently on a separate thread while the mappers do partial decision.

Another solution can be to make partial decisions within the mappers during the periodical update for all service functions and then ranking them in max-heaps; each max-heap ranks locations by single service function. The partial decision will be passed to the reducer which will decide from the pre-processed partial decisions. This solution saves time, especially with large search space, since the reducer will not wait for mappers to produce results. However, this solution will need an adjustment to the MapReduce implementation, for example, the Mapper will only calculate a similarity function for the time and risk, but the number of locations per application is done by Reducer.

5.4.2 Evaluating GNH's Optimisation Objectives

In this section, we evaluated the qualities of GNH solutions in terms of completion time, failure rate, and cost. During the test, we compared the GNH results with two approaches: random-based and PSO-based approaches. Redundant deployment was considered in both approaches. Outlining the experiment setup, followed by the results of the experiments.

Experiment Setup

In this experiment, 10,000 application requests, uniformly dispersed over 24 hours, are sent to a fog node controlling 100 locations within an IoT, fog, and cloud ecosystem. The synthetic data generator (Section 4.4.1) and the MTBF clock (Section 4.4.2) are used for simulation data and failure timings respectively.

Variable	Number/Rangs
Application requests	10,000
SFC length	(1-20)
Location	100
FNs	80
VMs	20
FN's Latency	(21 - 50 ms)
VM's Latency	(50 - 300 ms)
FN MTTF	(10 - 30 ms)
FN MTTR	(5 - 15 ms)
VM MTTF	(30 - 300 ms)
VM MTTR	(2 - 10 ms)

Table 5.1: The simulation parameters are chosen randomly from these ranges.

To simulate variable connection conditions, a random network delay is introduced. Specific delay ranges for FNs and VM instances, reflecting the variability in network conditions, are provided in Table 5.1. Each deployment scenario selects a delay value from these ranges based on a uniform distribution. The types and hardware capabilities of FNs and VMs are detailed in Table 5.2 and Table 5.3, respectively.

The GNH method's evaluation occurs within an infrastructure where nodes experience frequent failures. Related parameters for this failure model are specified in the MTBF clock, Table 5.1 (Section 4.4.2 in *Chapter 4* for MTBF clock algorithm).

[H] **Version**	**CPU**	**Core(s)**	**Memory**			**Storage**			**Network Interface Speed**	
RPi 3 Model A+	1.4 GHz	4	256 MB	512 MB		8 GB	16 GB	32 GB	300 Mbps	
RPi 1 Model B	700 MHz	1	256 MB	512 MB		8 GB	16 GB	32 GB	100 Mbps	
RPi 1 Model B+	700 MHz	1	256 MB	512 MB		8 GB	16 GB	32 GB	100 Mbps	
RPi 2 Model B	900 MHz	4		1 GB		8 GB	16 GB	32 GB	100 Mbps	
RPi 3 Model B	1.2 GHz	4		1 GB		8 GB	16 GB	32 GB	100 Mbps	300 Mbps
RPi 3 Model B+	1.4 GHz	4		1 GB		8 GB	16 GB	32 GB	300 Mbps	1000 Mbps
RPi 4 Model B	1.5 GHz	4	1 GB	2 GB	4 GB	8 GB	16 GB	32 GB	300 Mbps	1000 Mbps
RPi Zero W	1 GHz	1		512 MB		8 GB	16 GB	32 GB	300 Mbps	

Table 5.2: Variety of raspberry pi (RPi) models choose from

[H] Version	CPU		Core(s)	Memory	Storage	Network Interface Speed	Max NICs
VM 1	2.35 GHz	3.35 GHz	2	8 GB	50 GB	1000 Mbps	2
VM 2	2.35 GHz	3.35 GHz	4	16 GB	100 GB	2000 Mbps	2
VM 3	2.35 GHz	3.35 GHz	8	32 GB	200 GB	2000 Mbps	4
VM 4	2.35 GHz	3.35 GHz	16	64 GB	400 GB	2000 Mbps	8
VM 5	2.35 GHz	3.35 GHz	32	128 GB	800 GB	16000 Mbps	8

Table 5.3: Possible virtual machines (VMs) that are chosen from

Service requirements	**Max Value**
CPU	2,000,000
Memory	6MB
Storage	5MB

Table 5.4: Maximum computational resource requirements of the generated functions.

We compare the GNH approach with three alternative methods: simple *Greedy* approach (no replica), a random allocation approach (*Rand*) and Particle Swarm Optimisation (*PSO*). The methods are assessed for deploying service functions with and without replicas, with the exception of the *Greedy* approach that applies ILP but excludes $MaxReplica_{i,j}$. In this context, *RPSO* refers to the *PSO* method with two replicas, while *RP* denotes the random placement method with a maximum number of replicas as defined by Formula 3.3 in *Chapter 3*. PSO was selected for comparison due to its operational similarity with GNH, as both methods generate a pool of candidate solutions before choosing the optimal one. The parameters used in these experiments are detailed in Tables 5.1 and 5.4.

Results

The heatmap in Figure 5.3 shows application completion times. We see that 63.46% of *GNH* allocations take less than 100 ms. Whereas 29.33% and 5.31% of applications finished within 101ms-200ms and 201ms-500ms, respectively. The 29.33% applications that have longer completion times are due to the application having longer chained service functions. Around 1.6% of the applications fail to complete with low delay.

The heatmap shows the Greedy algorithm completes 54.46% of applications in under 100ms, with 29.18% within 101-200 ms, indicating quick task processing. Performance remains acceptable for 8.51% of applications completed between 201-500 ms. However, with replicas, GNH achieves a 63.46% completion rate for tasks under 100ms, bettering the Greedy algorithm's 54.46%, showcasing more efficient and rapid task processing capabilities.

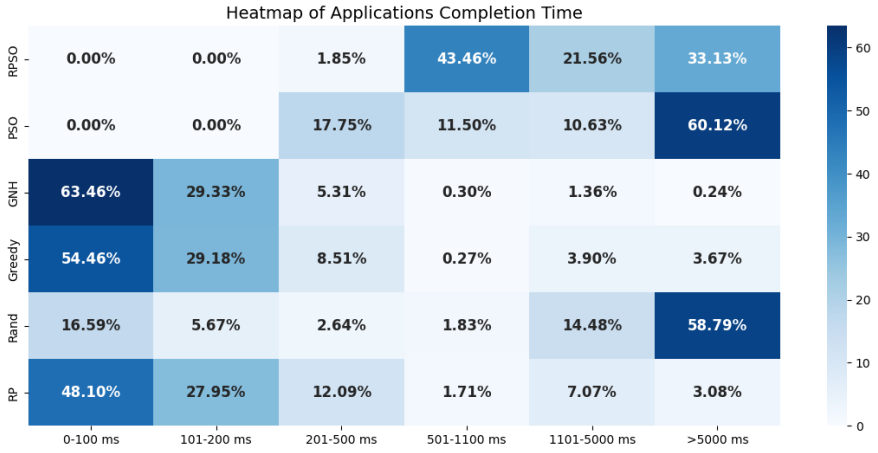


Figure 5.3: Heat-map shows applications completion time

Compared to GNH, Particle Swarm Optimisation with Replicas (RPSO) completed most of its applications within 5001-1100 ms. This is because RPSO takes more time to decide where to deploy applications; the PSO-based decision-making process takes 200-300 ms before deploying the application (Table 5.5) with 5 particles and 50 iterations.

	No Replicas (*seconds*)				**With 2 Replicas** (*seconds*)			
Iteration	**50**	**100**	**200**	**400**	**50**	**100**	**200**	**400**
Particles								
5	0.238	0.486	1.002	1.985	0.343	0.611	1.255	2.442
10	0.849	1.698	3.397	6.802	0.972	1.953	3.884	7.665
20	3.110	6.166	12.332	24.717	3.349	6.670	13.323	26.709
40	11.910	23.855	47.010	93.905	12.548	24.900	49.874	99.550

Table 5.5: Comparing PSO performance - SFC length is 10

Particle Swarm Optimisation without replica (PSO) did not perform well compared to either GNH or RPSO. Around half of the applications failed to complete on time, and 42.46% of the deployed applications completed between 500-1100 ms.

Increasing the number of particles also increases the chances of converging to the global optimum. However, more particles and iterations will increase execution time, as shown in Table 5.5. For example, in a Raspberry Pi 3B+, 10 particles with 250 iterations can reduce failure rate. However, it will take between 3.5 to 4.5 seconds to complete the 250 situations.

Using *RP*, 76.05% applications completed in less than 200ms. Approximately 16.74% of applications can be allocated to faster locations, if the *RP* was aware of location completion times. Therefore, even though the replica-based strategy mitigates failure, on its own it will not guarantee an optimal completion time. Finally, since *Rand* does not use replicas, it is more prone to failure when compared to *GNH* and *RP*. More than 70% of *Rand* allocations finished in the order of seconds not milliseconds. The Greedy method, using ILP without $MaxReplica_{i,j}$, completes 92.15% of workflows in under 500ms, outperforming RP's reliance on $MaxReplica_{i,j}$ by about 4%.

Figure 5.4 shows the average completion time of applications and their execution time. *GNH* on average completes the application request in less than 200ms. Whereas *RP*

has an average completion time of around 540ms.

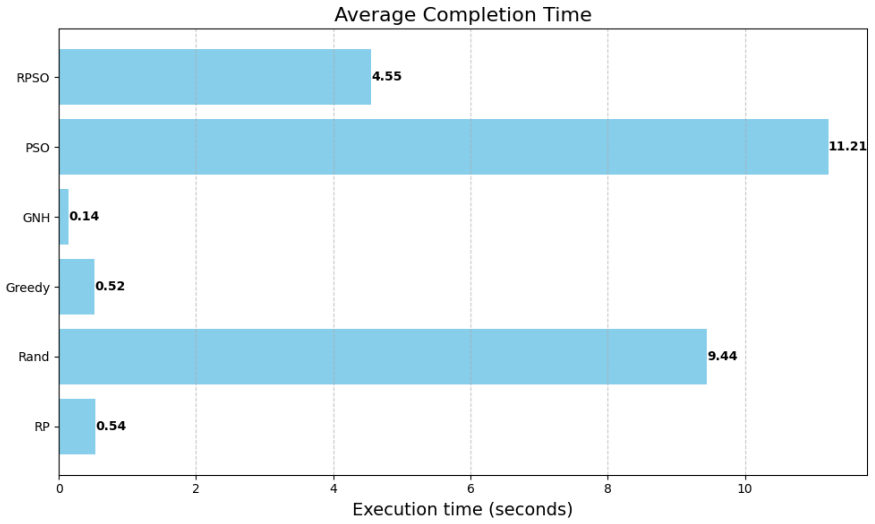


Figure 5.4: Algorithm comparison: completion time in seconds

Table 5.6 shows the failure percentage for each algorithm. When using *GNH*, there is a 3.15% failure rate, which is the lowest among the presented algorithms. *Greedy* has a failure rate of 8.83%, significantly better than most other algorithms listed, yet not as low as *GNH*. The failure rate for *RP* stands at 17.23%, indicating that this percentage of the allocated applications fail to complete within the expected time. With *Rand*, we observe a high failure rate of 75.77%, which means only about 24.23% of the allocated applications succeed in the expected time. In comparison, *PSO* has a failure rate of 72.59%, which is a slight enhancement over *Rand*, increasing the success rate by about 3.18%. *RPSO* has a failure rate of 45.64%, which represents a significant improvement over *Rand*, reducing the failure rate by approximately 30.13%.

Algorithm	Failure rate
GNH	3.15%
Greedy	8.83%
RP	17.23%
RPSO	45.64%
PSO	72.59%
Rand	75.77%

Table 5.6: Each algorithm's failure percentage (on average)

The bar chart data, in Figure 5.5, shows resource utilisation for three categories: RPSO at 6.79 locations, GNH at 5 locations, and RP at 18.77 locations.

Finally, Figure 5.6 shows the *Cost* of application deployment for *RP*, *GNH*, and *RPSO* in hex-bin plots. GNH exhibits a tightly clustered usage of just above five locations per application. In contrast, RP shows a broader distribution, utilising up to approximately 50 locations for some application sequences. RPSO demonstrates an average use of locations that is greater than GNH but does not reach the upper distribution levels of RP. Although RPSO typically uses more than the base number of locations required by the SFC length, in rare instances, it can reach up to 25 locations per application. This indicates that while RPSO is generally efficient, certain application deployments may necessitate a significantly higher number of locations.

GNH varies the number of replicas based on the impact of specific service functions on the SFC completion time, and replicas are all deployed at the same time. It finds the replicas by dividing process nodes, and searching each in parallel to speed up the process. Moreover, the optimal allocation (i.e., local optimum) is guaranteed since all nodes are covered and ranked. GNH is deterministic, which means with the same environmental condition and the same input it will generate the same output.

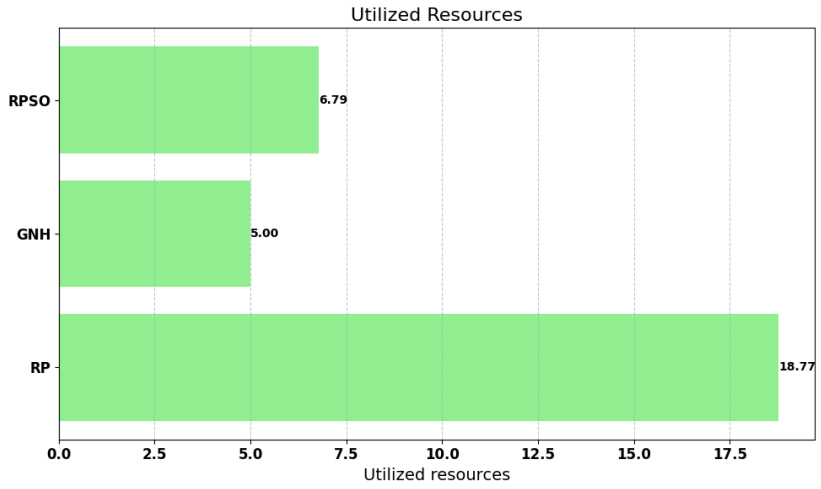


Figure 5.5: Average cost – based on the number of locations used

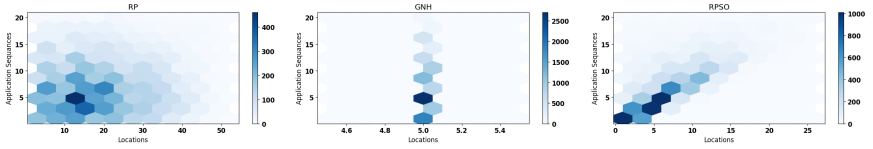


Figure 5.6: Relation between SFC length and the number of locations used by each algorithm is shown in the hex-bins chart .

5.5 Conclusion

This chapter describes the Greedy Nominator Heuristic (GNH), a greedy method that uses the MapReduce paradigm to reduce end-to-end latency across an SFC. Using the formual from *Chapter 3*, GNH applies two key strategies: (i) avoiding placement of functions on unstable computing resources, that is, resources that historically have demonstrated a high failure rate; and (ii) deploying functions across multiple locations, using a replication strategy that takes into account the location of the function in the SFC. Functions that occur at an early stage of the SFC have a greater replication factor, as successfully executing these functions has an impact on the completion of dependent functions further down the SFC.

We conducted a simulation-based evaluation of this work using parameters based on a Raspberry Pi deployment platform. The simulation is used to: (i) dynamically gen-

erate requests and vary the number of functions in an SFC (from 1 to 20); (ii) vary the availability and failure profile of resources using a clock mechanism that aligns resource unavailability with request arrival rate (using Mean-time-to-Failure and Mean-time-to-Recovery metrics). We create a number of possible simulation scenarios to compare GNH with two random placement algorithms, one with replicated placement of functions. On unreliable infrastructure, our results show that with the two strategies (i.e., redundancy and failure tracking), the system is able to reduce function execution latency by up to 68.38% compared to a redundancy only strategy. Moreover, the GNH redundancy is also shown to be cost-effective compared to a random redundant deployment.

GNH outperforms PSO for application deployment in fog-cloud infrastructure. PSO performance is less effective than the greedy strategy due to a number of factors, one of which is that the greedy technique is a dynamic optimisation approach by design. However, PSO is a static optimisation technique, and without the resetting iteration, it does not adapt to changes in decision variables. Also, every meta-heuristic algorithm, such as PSO, must take time to converge to a global optimal solution.

The next chapter *(Chapter 6)*, we will show how the greedy technique and meta-heuristics can be used together to make a better dynamic optimisation strategy. Therefore, the scheduling algorithm has the best of both worlds: fast scheduling and high-quality solutions.

Chapter 6

Enhanced Optimised Greedy Nominator Heuristic (EO-GNH): Enhancing GNH with Meta-Heuristics

6.1 Introduction

Chapter 5 presents GNH, which is a greedy approach that utilises MapReduce to speed up the scheduling process. Also, the chapter proves through experiments that GNH is better than PSO at scheduling application placements. GNH provides a fast local optimal placement plan. This chapter focuses on enhancing the quality of the scheduling solutions using *meta-heuristics*. Moreover, the chapter improves the MapReduce model to overcome meta-heuristic limitations.

The goal of this chapter is to extend GNH using *meta-heuristics*. The enhanced algorithm is as fast as GNH and provides a better placement plan from the SFC graph. The solution of the algorithm is *non-dominated* by other solutions, which means it is part of the *pareto front*. The solution outperforms the final solution across all objectives. The objectives are low delay, low cost, and low risk.

In scheduling real-time applications, *meta-heuristics* have to address three primary issues: (i) accelerating convergence to optimal solutions, as demonstrated with PSO in Chapter 5, (ii) providing a dynamic optimisation strategy that adapts to changes in the

environment, and (iii) determining the most suitable meta-heuristic for a given application [86] (also covered in the survey results in *Chapter 2*). To address these issues, we introduce a parallel model that not only speeds up meta-heuristics to overcome slow convergence but also adapts to environmental changes. Additionally, we develop a machine learning solution to assist in selecting the most suitable meta-heuristic for a specific application.

Asynchronous MapReduce overcomes slowness in meta-heuristics. Each Mapper is a meta-heuristic, which continuously refines the pareto front. When a function in the SFC graph is ready to be deployed, the Reducer chooses the best solution out of all pareto-front approximations provided by Mappers.

Using *machine learning* to forecast the most stable meta-heuristic for scheduling. The forecast is based on the features of the application and the infrastructure. The meta-heuristics are then prioritised based on the forecast results and submitted to mappers to initiate the scheduling process.

The remainder of this chapter is as follows: Section 6.2 provides research methodologies. Section 6.3 details the algorithm in depth. Section 6.3 evaluates the optimisation algorithm, compares it to another distributed algorithm, and puts the algorithm to the test in a simulated environment. Finally, Section 6.5 brings the chapter to a close.

6.2 Methodology

This section describes the research methodology to develop and evaluate the algorithm. First, it presents the requirements. Second, it sets up the evaluation criteria. Finally, it highlights the test environment.

6.2.1 Scheduling Requirements

The scheduling process must overcome GNH's limitations: it must escape local optima, provide non-dominant solutions, and be fast regardless of the mapper's setup. When re-optimising, the algorithm selects solutions from the pareto front, resulting in an optimally refined schedule.

Since scheduling involves meta-heuristics, it should be fast and provide solutions better than GNH's. Also, the algorithm's performance must be analysed to select the best meta-heuristic. This involves benchmarking meta-heuristics scheduling capability under different locations and SFC graph configurations.

Meta-heuristics outperform greedy approaches in terms of optimisation objectives outcome because they can escape from local optima [87]. However, *Chapter 5* shows that meta-heuristics take time to find a good solution, which affects the end-to-end latency of application execution. Therefore, it is necessary to overcome the slowness of meta-heuristics. This would provide better results than GNH in terms of scheduling speed and optimisation objectives.

Chapter 5 shows that the MapReduce technique is applied to accelerate scheduling by distributing decision variables across Mappers. However, the Reducer awaits the completion of all Mappers before returning the final results; this is synchronous execution. Given that meta-heuristics require more time than a greedy method, building an asynchronous MapReduce would accelerate the procedure. Therefore, the enhanced scheduling algorithm requires an asynchronous implementation of MapReduce.

6.2.2 Evaluation Criteria

Evaluation considers two aspects: *algorithmic execution efficiency* and *solution qualities*. Algorithm efficiency is related to *algorithm speed*, *algorithm memory overhead*, and *Pareto front volume* . Whereas solutions qualities is related to *makespan*, *risk of*

non-completion, and *cost*.

In the algorithmic execution efficacy, comparing the optimisation algorithm with a state-of-the-art distributed meta-heuristic approach using benchmark objective functions. The distributed meta-heuristic is distributed non-dominated sorting genetic algorithm II (dNSGA-II) [88], and benchmark objective functions is ZDT1 [89]. Periodically, we examine the quantity of non-dominant solutions to assess the quality of solutions over time.

The solution quality of the EO-GNH algorithm is compared to that of the *GNH*, *Greedy* (with no replica), Round Robin (*RR*), and Random Placement (*Rand*) algorithms, focusing on time, cost, and service availability. *RR* and *Rand* serve as the baseline algorithms in this comparison. This approach provides a detailed and precise view of EO-GNH's performance against both these standard benchmarks and the informed algorithms such as *GNH* and *Greedy*.

6.2.3 Test Environment

During run time, a snapshat of the pareto front and a reading of the memory overhead are taken periodically to determine the efficiency of the algorithm executed. This provides a profile of the pareto front size as well as the time and resources utilised overtime.

The experimental framework observes and evaluates the performance of our scheduling model. *Python time* [70] and *psutil* [72] are two Python modules used to evaluate the computational efficiency of the algorithm. The time library provides numerous time-related features that assist in timing the snapshat periods throughout run time. Psutil is a library of system and process tools that offers information about running processes and system utilisation.

6.3 EO-GNH Algorithm

The optimisation algorithm searches for solution out of the pareto front. EO-GNH employs MapReduce and meta-heuristics to identify potential locations for redundant deployments. Workers (i.e., Mappers) use meta-heuristics to search the decision variables. Mappers' results are shared with the Reducer (i.e., control fog node), whose results determine the best locations for redundant deployment across all. Mappers provide non-dominant solutions during the execution of the SFC graph. When the time comes to decide where to place a function, the Mappers results are accessed by the Reducer to generate a solution.

6.3.1 Algorithm Components

The **parallel model's** execution is asynchronisation-based, meaning that *Reducer* does not wait for all *Mappers* to complete their meta-heuristic iterations before starting to provide placement decisions. The Reducer has access to each iteration result from the Mappers. Each Mapper has a file to write the current optimal solutions of the meta-heuristic, and the Reducer has reading access to the Mappers' shared files. Once a Mapper produces a solution, regardless of the quality of the solution, the Reducer can consume it and provide a decision for single-function placement. *Parsl* [9] is utilised to enable MapReduce computing.

The Oracle in computer science is software that is queried for answers to specific questions [90]. The Oracle here has knowledge about match of meta-heuristics and application/infrastructure configuration. The query results are the best-suited meta-heuristics based on the objectives preferences, e.g., makespan, cost, and then risk. The variables of the query are Mappers number, SFC size, locations size, population size, and criteria preferences. Within the Oracle, decision tree models are used to provide an approximate answer based on the meta-heuristics' previous performance. Oracle has three stages: (i) infer decision trees, (ii) use preference-based sorting, and (iii)

assign meta-heuristics to Mappers. Figure 6.2 shows how the Oracle interacts with the MapReduce approach.

A **decision tree** is used to forecast the quality of a meta-heuristic. A tree can be viewed as a piecewise constant approximation [91]. The forecast result is the approximated objective function output of one meta-heuristic, such as NSGAII. The training dataset for models is comprised of meta-heuristics benchmark data. Benchmarking meta-heuristics under different SFC and location size set-ups, which are the selected features. Features and the output of the benchmark used to train the decision tree. The benchmark phase is done using a simulation. The simulation with the inputs, i.e., features, from table 6.1, then aggregated the output with the inputs creating the dateset to prepare for the training phase.

Attribute	Value		
SFC size	5	10	20
Location size	100	500	1000
population/swarm size	10	50	100

Table 6.1: Dataset used to train the decision tree was made up of the features chosen for the decision tree .

Meta-heuristics are heuristic algorithms that can be used to solve a wide range of different types of problems. We use some nature-inspired meta-heuristic algorithms, shown in Table 6.2, from the *jMetalPy* [69] library. During runtime, algorithms provide multiple solutions. Each set of solutions is a pareto-front approximation, i.e., the best front discovered. Usually, meta-heuristics are composed of main loops, with each iteration in them performing a step. Steps of the main loop where meta-heuristics have differences. For example, the step contains a simulation of a swarm for PSO-based, whereas Genetic Algorithm (GA)-based applies genetic operators. In the proposed parallel model, after each step of the meta-heuristics current discovery, the pareto front is saved in a shared file.

Algorithm	Meta-heuristic bases
GDE3	Genetic Algorithm
HYPE	Genetic Algorithm
IBEA	Genetic Algorithm
MOCell	Genetic Algorithm
NSGAII	Genetic Algorithm
OMOPSO	Particle Swarm Optimisation
SMPSO	Particle Swarm Optimisation

Table 6.2: List of meta-heuristics utilised by EO-GNH

Solution encoding is the method by which the meta-heuristic data handle the data representation for the solution. This includes the data's type and structure. In our system, the solution encoding is an array of integers, where the value is the location ID and the indices represent functions in the graph. As shown in Figure 6.1, multiple indices belong to one function, which holds the redundant placement plan for that function. In PSO-based algorithms, the solution result is of the float type (a real number), which is not an integer. We utilised discretisation methods to convert elements of the array into an integer. The mathematical floor function is used to remove decimal point.

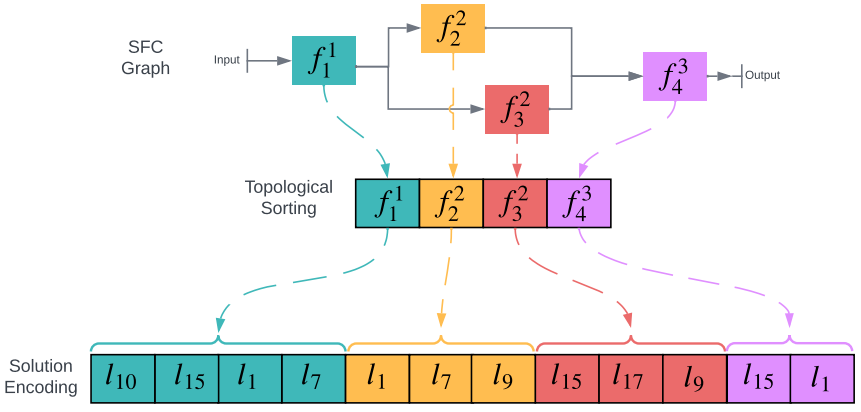


Figure 6.1: SFC redundant deployments solution encoding. The solution encoding's elements are location IDs, while an array's indices specify the function.

Timsort[1] is a hybrid, stable sorting algorithm built from merge sort and insertion sort

[1]Timsort has been the default sorting algorithm in Python since version 2.3.

that is optimised for a wide variety of data types. Used to sort decision tree outputs by user objectives and preferences, i.e., makespan, cost, and risk.

The **greedy** approach is employed in EO-GNH, making it an extension of the GNH. The greedy heuristics (i.e., Reducers) of EO-GNH process the output file of each meta-heuristic (i.e., Mappers). The planning for the next function deployment happens when the currently executed function is completed. Refer to *Chapter 4*, Figure 4.10, for the steps to 'iterate over the SFC.' During this planning, Reducers implement the greedy approach, utilising the similarity functions described in Formula 5.1 as heuristics.

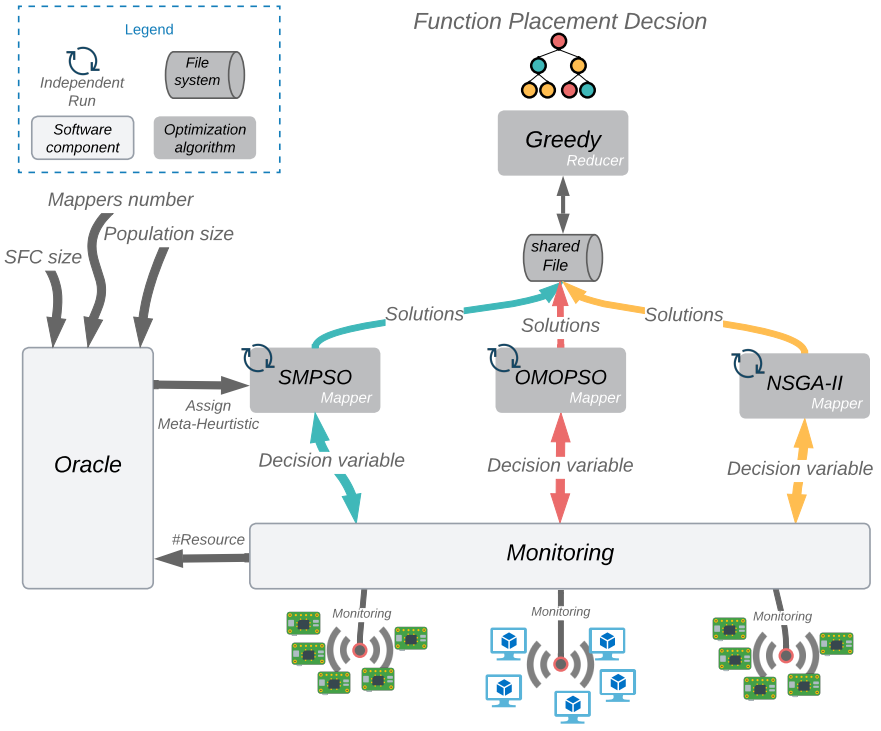


Figure 6.2: Asynchronous MapReduce performs EO-GNH, initiated by the Oracle. Each mapper is a meta-heuristic selected by the Oracle based on prior knowledge acquired during the training phase of its decision trees. The Oracle ranks meta-heuristic algorithms according to their attributes. The reducer heuristic is manually selected as greedy .

6.3.2 Algorithm Workflow

The EO-GNH's operational framework is based on Oracle and asynchronous MapReduce mechanisms. The algorithm's workflow consists of three major phases: (i) contacting the Oracle, (ii) initialising and activating the meta-heuristics, and (iii) operating the application while simultaneously adapting to changes. The following sections will elaborate on these essential phases, which form the process's foundation.

The Oracle Workflow

The Oracle is designed to identify the optimal meta-heuristics for running an application, given the parameters of the underlying infrastructure. As illustrated in Figure 6.3, the query is formed from various attributes, including the controller's capacity, algorithm parameters, location data, the SFC graph, and user objective preferences. These user preferences guide the prioritisation of the output.

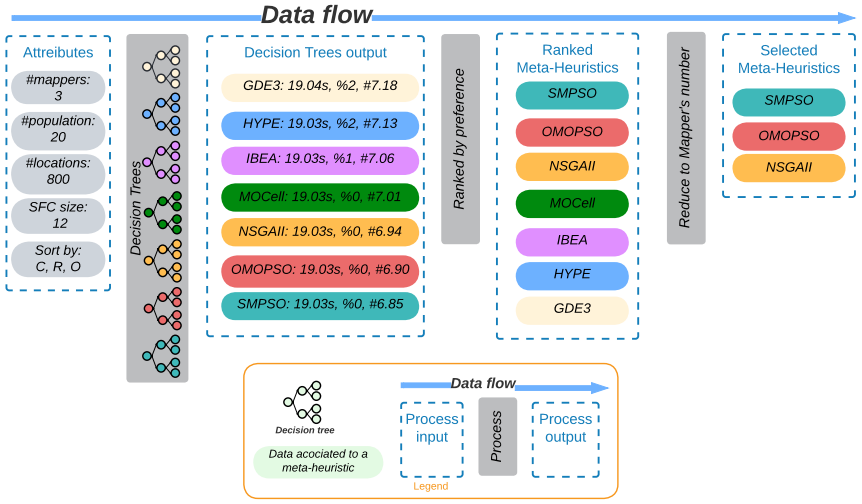


Figure 6.3: The Oracle ranks and selects meta-heuristics, each has a color. Inputs for the oracle are the number of SFC mappers, population, locations, and functions. The ranking is based on the makespan (C), the risk (R), and the number of locations utilised (O) .

To estimate the expected results of each meta-heuristic, decision tree models are used.

These models generate predictions on key factors like makespan, risk, and cost. Subsequently, the meta-heuristics are evaluated and ranked in alignment with user preferences. Lastly, the highest-ranked meta-heuristics are assigned to the available mappers, thus concluding the process.

Meta-heuristic Workflow

Contrasting with the Mappers in GNH, EO-GNH makes decisions involving the entirety of the SFC graph, as represented in Figure 6.1. This decision-making process is detailed in Algorithm 6, which demonstrates how the Mapper initiates the meta-heuristic (line 3), proceeding to enter the main cycle.

Within each cycle, the decision variables undergo an update (lines 5-7), leading to a concurrent adjustment of the current objective values (outputs of the objective functions). The *Step* procedure utilises the previous solution, in conjunction with locations and the SFC, to carry out a singular step.

This method subsequently yields a new solution, saved within a shared file accessible to the Reducer (lines 9-10). The algorithm halts once the execution of the SFC is concluded (indicated by the NOTCOMPLETED value in line 4 becoming *False*)

The *Step* method, referenced in line 8, incorporates operations specific to meta-heuristics. For derivative algorithms stemming from PSO, operations are enacted in this order: (i) velocity update, (ii) position update, (iii) objectives evaluation, (iv) global best update, and (v) particle best update.

On the other hand, GA-based procedures involve: (i) selection, (ii) crossover, (iii) mutation, (iv) objectives evaluation, and (v) replacement. Notably, in the GA derivation, the replacement process updates the solution while preserving the superior offspring, further enhancing the evolutionary progress.

Algorithm 6 The $solution$ is an array where each index refers to the function (f_j^i), whereas its content is the location id

```
1: class MAPPER
2:     method MAP (L; SFC)
3:         solution ← INITSOLUTION( L; SFC )
4:         while NOTCOMPLETED do
5:             if IsDecisionVariableChanges(L; SFC): then
6:                 DecisionVariable ← UPDATEDECISIONVARIABLE(L; SFC)
7:                 solution ← UPDATESOLUTION(solution)
8:             solution ← STEP(L; SFC; solution)
9:             MapperResult ← SOLUTION
10:            solution ← UPDATESHAREDFILE(MapperResult)
```

6.4 Evaluation

This section evaluates the EO-GNH's performance. First, this section evaluates efficiency performance. Efficiency performance focuses on the parallel model running the meta-heuristics. Second, it assesses the quality of the solutions provided.

6.4.1 Efficiency Performance Evaluation

In this section, we conduct an evaluation of EO-GNH's performance in comparison with the distributed Non-dominated Sorting Genetic Algorithm II (dNSGAII) [88]. Notably, both of these algorithms operate asynchronously. Moreover, instances of NSGAII have been utilised within the EO-GNH framework as Mappers and Reducers, enabling us to assess the number of non-dominated solutions they produce.

Setup Experiment

In this assessment, we explore parallel optimisation strategies using ZDT1, a synthetic test problem, as a benchmark [89]. Both algorithms utilise Python tools that facilitate parallelism. Specifically, EO-GNH employs Parsl [73], while dNSGAII utilises Apache Dask [92]. To optimise the extraction of non-dominant solutions, the EO-GNH

reducer is configured to gather the Pareto fronts from the mappers. Additionally, periodic snapshots of the Pareto front are taken to evaluate the algorithms' performance over time.

The experiment is conducted on Google's cloud servers, within a Jupyter notebook environment (i.e., Google's Colaboratory). The virtual machine used is equipped with a single-core Intel(R) Xeon(R) CPU operating at 2300 MHz, coupled with 12 GB of RAM, and operates without a GPU. Both Dask and Parsl are designed to maximise utilisation of two cores; Dask accomplishes this directly, while Parsl assigns an executor to each core.

dNSGAII implements NSGAII operators, such as selection and reproduction, asynchronously. Each stage of the dNSGAII collects the outcomes of the completed operation, passing them onto the subsequent one - for instance, delivering selection results to the reproduction process. Despite the asynchronous nature of dNSGAII, its iterations maintain synchronisation.

Contrarily, EO-GNH deploys each NSGAII as an independent algorithm, with workers operating asynchronously while a master node regularly collates the current solutions. When the system has additional resources, EO-GNH scales up, deploying more instances of NSGAII to enhance performance outcomes.

Results

Ten seconds into the operation, EO-GNH is observed to amass a larger number of solutions that can be utilised by the controller. Additionally, the quality of the Pareto front approximated by EO-GNH presents significant advantages, as depicted in Figure 6.4. Notably, EO-GNH manages to match the 10-second performance of dNSGAII within just two seconds.

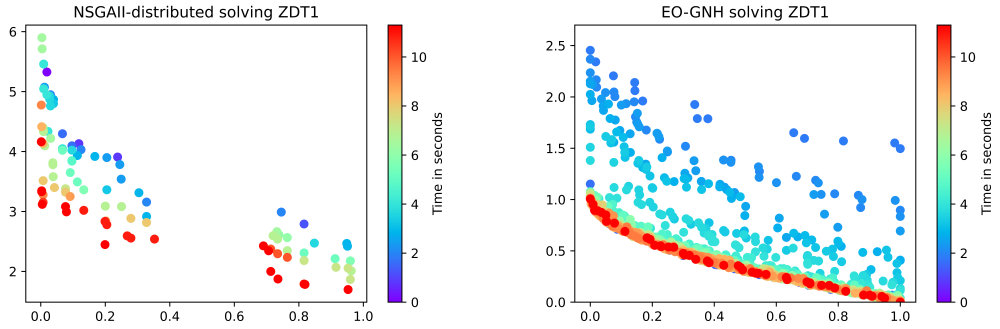
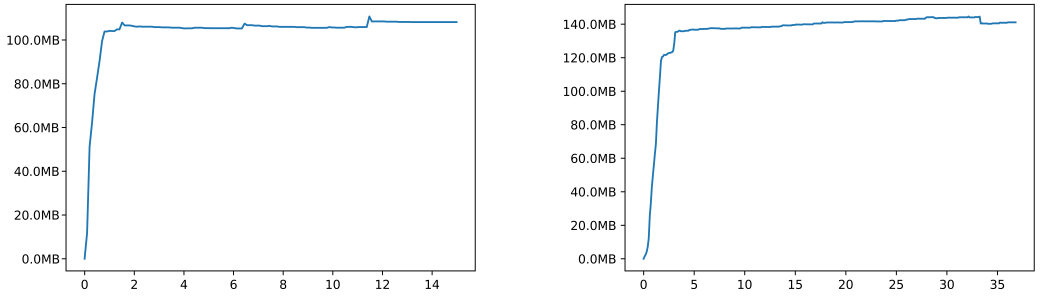


Figure 6.4: Algorithm comparison: when solving the *ZDT1* problem, the colour is assigned to the time the pareto front was collected .

However, despite the asynchronous nature of dNSGAII, its iterations are synchronous. This operational characteristic results in fewer non-dominant solutions being captured, as demonstrated in Figure 6.4. Further, the deployment of NSGAII operations is often subjected to delays due to queue waiting time.

In contrast, EO-GNH operates multiple instances of NSGAII across two different processes, thereby reducing the likelihood of a process being left idle waiting for CPU time. This strategic implementation results in more efficient utilisation of computational resources and helps to drive improved performance.



EO-GNH dNSGAII

Figure 6.5: Memory overhead overtime for the distributed algorithms

Figure 6.5 displays the memory overhead associated with running the algorithms over

time, measured in seconds. Interestingly, there are observable differences in the memory usage patterns of EO-GNH and dNSGAII.

Initially, EO-GNH's memory usage saw a sharp increase from 0 to 60MB in less than a second. Conversely, dNSGAII demonstrated a slower rate of memory consumption, taking around 2 seconds to reach the same level of 60MB.

Following this initial phase, the EO-GNH's memory usage presented fluctuations between 100MB and 110MB. This pattern was sustained throughout the duration of the algorithm's iterations. On the other hand, dNSGAII's memory consumption exhibited a different behaviour, displaying a more gradual increase from 120MB to 140MB after the third second. This level of memory usage was then maintained consistently throughout the remaining iterations.

During a single run, it appears that both EO-GNH and dNSGAII utilised a fixed amount of memory. For EO-GNH, there is a possibility that the memory overhead may increase if more instances of NSGAII are added. In contrast, dNSGAII's memory usage would likely remain stable unless additional computing cores are incorporated, a scenario that Dask can handle effectively.

In essence, EO-GNH leverages Parsl to add an extra degree of flexibility, allowing it to dynamically use more resources when available. Conversely, dNSGAII is designed to utilise all available resources from the outset, thereby maintaining a constant memory usage regardless of additional resources.

6.4.2 Evaluating EO-GNH's Optimisation Objectives

Setup Experiment

In this experimental setup, we evaluate the GNH and EO-GNH algorithms within a fog infrastructure comprising Fog Nodes (FNs), simulated by Raspberry Pi 4B units. The scenario involves handling 4000 application execution requests across 800 locations.

Variables	Number/Ranges
Application requests	4000
SFC sizes	(4, 8, 12, 16)
SFC types	400
Packet sizes	(1KB - 1MB)
population/ swarm size	20
FNs	800
MTTR	(5- 15s)
MTTF	(10 - 30s)

Table 6.3: The simulation parameters are chosen randomly from these ranges

The experiment simulates 4000 application requests dynamically invoked. These requests operate within the constraints defined in Table 6.3. To emulate the variability of resource availability and failure profiles, we employ a clock mechanism. This mechanism alternates between simulating a failure event with a Mean Time to Repair (MTTR) and regular resource operations characterised by a Mean Time to Failure (MTTF).

For a comprehensive performance overview, we compare GNH and EO-GNH with random placement (RP), a simple Greedy algorithm, and Round-Robin load balancing (RR). Furthermore, the study evaluates the performance of various meta-heuristics serving as Mappers. These meta-heuristics include GDE3, HYPE, IBEA, MOCell, NSGAII, OMOPSO, and SMPSO. This comparison allows us to examine the effectiveness and efficiency of these methods under different configurations involving 1, 2, 3, and 4 Mappers.

We generate application request arrival times and workflow compositions as records. Each workflow deployment is subject to a user-defined deadline, which defines the expected completion time.

During the evaluation phase, we compare the anticipated and actual completion times to deployed states. A workflow deployment that does not meet its deadline is marked as a failure. We randomly select a failure model parameter for each FN, based on the clock mechanism described in Section 4.4.2. The link delay is estimated based on the packet size and a randomly assigned quality variable that affects transmission time.

We uniformly distribute application requests throughout a single day to emulate data streams generated by infrastructure controllers. This approach closely replicates the production of data streams.

Results

This section provides a detailed analysis of the simulation results, comparing the EO-GNH with the GNH and benchmark algorithms. The focus lies on studying EO-GNH's performance across different configurations with Mapper numbers ranging from 1 to 4. Additionally, we examine the EO-GNH's functioning with varied Mappers setups.

SFC size	EO-GNH-2	GNH	Greedy	Rand	RR
4	5.21	5.23	6.70	12.80	13.67
8	12.87	12.90	15.09	27.77	25.96
12	19.03	19.04	22.33	41.59	41.97
16	27.16	27.21	32.59	56.95	56.90

Table 6.4: Average makespan when there are 800 locations and a population of 20 .

Table 6.4 reveals that EO-GNH-2 (i.e., EO-GNH with 2 mappers) consistently outperforms the other algorithms across all SFC sizes. For SFCs of size 4, EO-GNH-2 completes tasks in an average of 5.21 seconds, which is slightly quicker than GNH at 5.23 seconds, and significantly faster than the Greedy, Rand, and RR methods, which require 6.70, 12.80, and 13.67 seconds, respectively.

This trend continues as the SFC size increases. For SFCs of size 8, EO-GNH-2's average makespan is 12.87 seconds, outperforming GNH (12.90 seconds), Greedy (15.09 seconds), Rand (27.77 seconds), and RR (25.96 seconds).

At an SFC size of 12, EO-GNH-2 remains the fastest with an average makespan of 19.03 seconds, slightly quicker than GNH's 19.04 seconds, and significantly faster

than Greedy (22.33 seconds), Rand (41.59 seconds), and RR (41.97 seconds).

For SFCs of size 16, EO-GNH-2's superior performance continues, recording an average makespan of 27.16 seconds. This result beats GNH's 27.21 seconds, Greedy's 32.59 seconds, and significantly outperforms Rand's 56.95 seconds and RR's 56.90 seconds.

SFC size	EO-GNH-2	GNH	Greedy	Rand	RR
4	%100	%98	%64	%16	%14
8	%100	%97	%39	%3	%4
12	%100	%97	%23	$\% \approx 0$	$\% \approx 0$
16	%100	%90	%3	$\% \approx 0$	$\% \approx 0$

Table 6.5: Successful rate when the number of locations is 800 and the population size is 20 .

The success rate is a crucial performance indicator, demonstrating the reliability and effectiveness of each algorithm. Table 6.5 shows the successful execution rate of various algorithms.

In Table 6.5, regardless of the SFC size, EO-GNH-2 exhibits a 100% success rate. This implies that it can consistently and reliably complete all tasks, offering a significant advantage over the other algorithms.

For SFCs of size 4, GNH and Greedy have respectable success rates of 98% and 64% respectively. However, the success rates of the Rand and RR algorithms are significantly lower, at 16% and 14%, respectively. As the SFC size increases to 8, the performance gap between EO-GNH-2 and the other algorithms becomes even more noticeable.

The success rates of GNH and Greedy drop to 97% and 39%, respectively. Moreover, Rand and RR see their success rates plummet to 3% and 4%, respectively. At an SFC

size of 12, GNH manages to maintain a 97% success rate, but Greedy's success rate falls drastically to 23%. Both Rand and RR's success rates fall to approximately 0%.

When dealing with the largest SFC size of 16, EO-GNH-2 remains consistent with a 100% success rate. In contrast, GNH sees its success rate drop to 90%, and Greedy's success rate plummets to a mere 3%. Again, Rand and RR fail to register a measurable success rate, hovering at approximately 0%.

SFC size	EO-GNH-2	GNH	Greedy	Rand	RR
4	4.59	5.36	1.53	3.99	4.0
8	5.54	6.14	2.04	7.96	8.0
12	6.83	7.38	2.72	11.93	12.0
16	8.54	9.01	3.76	15.87	16.0

Table 6.6: Average location is used, when the number of locations is 800 and the population is 20 .

The efficient use of locations can contribute to the effective management of resources and network load, as well as overall system performance and responsiveness. Table

6.6 illustrates the average number of locations utilised by various algorithms.

For an SFC size of 4, EO-GNH-2 has an average usage of 4.59 locations, which is the lowest among all algorithms, apart from Greedy, which uses only 1.53 locations on average. This low usage could be attributed to the Greedy algorithm's simplified approach, but it might not necessarily indicate better overall performance or a more balanced system load. In comparison, the GNH, Rand, and RR algorithms use more locations, with averages of 5.36, 3.99, and 4.0 respectively.

With an SFC size of 8, the EO-GNH-2 algorithm continues to be more efficient in terms of location usage, averaging at 5.54 locations. Again, the Greedy algorithm uses fewer locations, with an average of 2.04, while GNH, Rand, and RR use 6.14, 7.96, and 8.0 locations respectively.

In the scenarios with SFC sizes 12 and 16, the pattern continues. For size 12, EO-GNH-2 uses 6.83 locations on average, compared to GNH (7.38), Greedy (2.72), Rand (11.93), and RR (12.0). For size 16, EO-GNH-2 uses 8.54 locations on average, compared to GNH (9.01), Greedy (3.76), Rand (15.87), and RR (16.0).

SFC size	EO-GNH-1	EO-GNH-2	EO-GNH-3	EO-GNH-4
4	5.21	5.21	5.21	5.21
8	12.87	12.87	12.87	12.87
12	19.03	19.03	19.03	19.03
16	27.16	27.16	27.16	27.16

Table 6.7: Average makspan of EO-GNH with different mappers set up

Tables 6.8 and 6.9 collectively present a comprehensive overview of the EO-GNH algorithm's performance, considering a range of mapper configurations. Our analysis centres on two key metrics: the average makespan, which quantifies the total duration of task completion, and the average count of utilised locations, observed across a variety of SFC sizes.

SFC size	EO-GNH-1	EO-GNH-2	EO-GNH-3	EO-GNH-4
4	4.60	4.59	4.59	4.58
8	5.56	5.54	5.53	5.53
12	6.84	6.83	6.82	6.82
16	8.53	8.54	8.53	8.53

Table 6.8: Average used location by EO-GNH with different mapper number

In considering Table 6.9, it is evident that the average makespan of EO-GNH remains consistent, irrespective of the quantity of mappers deployed for all SFC sizes. Specifically, for SFC sizes of 4, 8, 12, and 16, the makespan is 5.21, 12.87, 19.03, and 27.16 seconds respectively, regardless of whether 1, 2, 3, or 4 mappers are utilised. This

indicates that the total duration of task completion within the EO-GNH configuration is not affected by the number of mappers.

SFC size	EO-GNH-1	EO-GNH-2	EO-GNH-3	EO-GNH-4
4	5.21	5.21	5.21	5.21
8	12.87	12.87	12.87	12.87
12	19.03	19.03	19.03	19.03
16	27.16	27.16	27.16	27.16

Table 6.9: Average makspan of EO-GNH with different mappers set up

Shifting our focus to Table 6.8, it becomes apparent that the average quantity of utilised locations experiences a slight variation dependent on the number of mappers implemented, though the alteration is not significantly impactful. For an SFC size of 4, the average quantity of utilised locations declines from 4.60 with a single mapper to 4.58 with four mappers. Similarly, for an SFC size of 8, the number drops from 5.56 to 5.53 as the quantity of mappers rises from one to four. This pattern is consistent for SFC sizes 12 and 16, displaying a minor decrease in utilised locations as the quantity of mappers increases.

Moreover, an examination of the success rates corresponding to the different mapper configurations reveals that all setups sustain a 100% success rate, with one exception. This anomaly is identified in the setup with a single mapper when deploying SFCs comprising 16 functions, which documents a marginally lower success rate of 99%.

SFC size	GDE3	HYPE	IBEA	MOCell	NSGAII	OMOPSO	SMPSO
4	5.21	5.21	5.21	5.21	5.21	5.21	5.21
8	12.88	12.87	12.87	12.87	12.87	12.87	12.87
12	19.04	19.03	19.03	19.03	19.03	19.03	19.03
16	27.20	27.19	27.18	27.17	27.17	27.17	27.17

Table 6.10: Makspan of meta-heuristics

Tables 6.10, 6.11, and 6.12 demonstrate the comparative performance of various meta-heuristics, including GDE3, HYPE, IBEA, MOCell, NSGAII, OMOPSO, and SMPSO, across a variety of measures: the makespan, success rate, and the cost based on the number of used locations, respectively.

SFC size	GDE3	HYPE	IBEA	MOCell	NSGAII	OMOPSO	SMPSO
4	%100	%100	%100	%100	%100	%100	%100
8	%99	%99	%100	%100	%100	%100	%100
12	%98	%98	%99	%100	%100	%100	%100
16	%92	%95	%96	%97	%97	%99	%99

Table 6.11: Successful rate of meta-heuristics

Table 6.10 illustrates the average makespan across different SFC sizes for all seven meta-heuristics. Across all SFC sizes, the makespan results are nearly identical. For SFC sizes of 4, 8, 12, and 16, the makespan falls in the ranges of 5.21, 12.87-12.88, 19.03-19.04, and 27.17-27.20 seconds respectively, with the variation between the heuristics being extremely marginal.

SFC size	GDE3	HYPE	IBEA	MOCell	NSGAII	OMOPSO	SMPSO
4	5.11	4.92	4.79	4.70	4.65	4.62	4.61
8	5.95	5.88	5.78	5.69	5.64	5.61	5.58
12	7.18	7.13	7.06	7.01	6.94	6.90	6.85
16	8.66	8.57	8.53	8.44	8.45	8.45	8.49

Table 6.12: Cost based on meta-heuristics used locations

In Table 6.11, we see the success rates for the seven meta-heuristics. For an SFC size of 4, all algorithms maintain a 100% success rate. As the SFC size increases, there is a slight decline in the success rate for GDE3, HYPE, IBEA, and even for MOCell when

the SFC size reaches 16. However, the decline is very modest, and all the algorithms maintain a high success rate, with all achieving at least 92% for an SFC size of 16.

Table 6.12 presents the cost based on the number of used locations for the meta-heuristics. Here, a clear pattern emerges: as the SFC size increases, so does the average number of used locations for all the algorithms. Interestingly, with an increase in the SFC size, a consistent downward trend in the number of used locations can be observed for all heuristics. This might imply a more efficient usage of locations as the complexity of the task (represented by the SFC size) increases.

Our enhanced framework (which improves on the findings mentioned in *Chapter 5*) adopts a dual-layer search approach similar to GNH by integrating asynchronous MapReduce with meta-heuristics to accelerate performance. Our framework addresses the slow convergence issue of meta-heuristic approaches like PSO. After every iteration, it dynamically updates the best solutions on the Pareto front, in contrast to the traditional scalarisation method used by greedy approaches. The framework delivers fair results across various meta-heuristic algorithms (such as GD3, HYPE) even without relying on an oracle (shown in Figure 6.6). The asynchronous approach significantly reduces decision-making time, addressing the latency issues identified earlier, making it an effective solution for optimisation challenges where latency is a critical performance factor.

In a direct comparison of execution times, the EO-GNH variants, particularly EO-GNH-4, with an average of 29.55 seconds, significantly excel over baseline algorithms like Random Placement (61.53 seconds) and Round Robin (60.08 seconds), reducing the time by around 32 seconds. Against GNH, which averages 29.60 seconds, EO-GNH-4 achieves a slight yet effective improvement of about 0.05 seconds. Among advanced meta-heuristics, EO-GNH-4 maintains a lead, outperforming SMPSO and OMOPSO, both averaging around 29.56 seconds, by approximately 0.01 seconds. This comparison highlights the efficiency of EO-GNH-4 in optimising execution times compared to both basic and more complex algorithms.

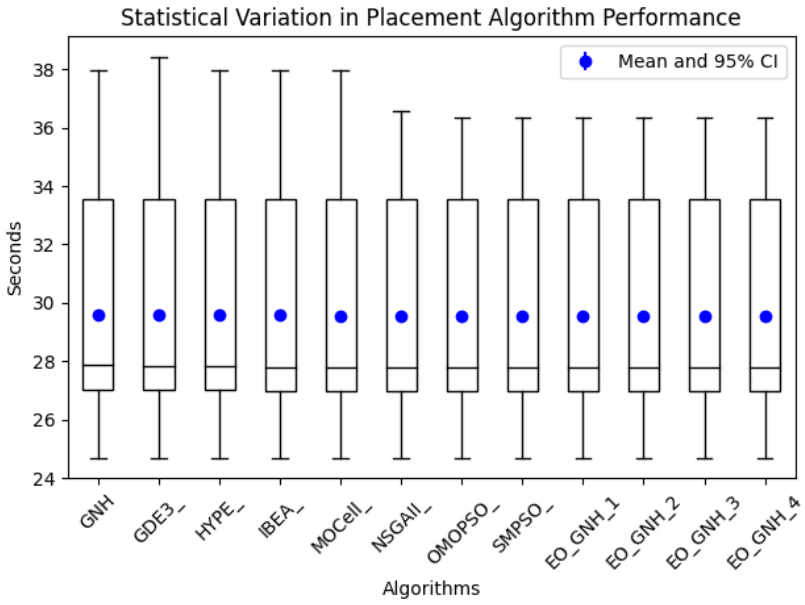


Figure 6.6: Boxplot comparison of execution times for placement algorithms, highlighting EO-GNH variants against established meta-heuristics..

The EO-GNH series demonstrates superior performance in managing requests exceeding 35 seconds, outshining both traditional and other meta-heuristic algorithms. Traditional approaches like Random Placement and Round Robin yield high delay rates of over 98%, whereas the Greedy algorithm reduces this to 44.63%. EO-GNH-3, however, significantly lowers delays to 7.6%, outperforming efficient alternatives such as GNH, GDE3, and HYPE. This efficiency makes EO-GNH-3 a prime choice for handling time-sensitive requests. In comparison to other meta-heuristics like GDE3 (8.1%), HYPE (8.0667%), IBEA (8.0%), MOCell (7.9%), NSGAII (7.7667%), OMOPSO (7.7%), and SMPSO (7.7%), EO-GNH's marginal advantage becomes crucial in scenarios demanding optimal time management and minimal processing delays.

The current experiment demonstrates that the addition of mappers will not have a direct impact on performance. However, *Chapter 7* examines the framework with a IoT application scenarios and provides a detailed description of how the integration of mappers can be utilised to enhance the performance of the framework.

6.5 Conclusion

This chapter describes the Enhanced Optimized-Greedy Nominator Heuristic (EO-GNH), an optimisation algorithm that uses asynchronous MapReduce to overcome GNH limitations. EO-GNH applies two key strategies: (i) selecting the most suitable meta-heuristics for scheduling an application; and (ii) running the greedy approach with meta-heuristics to speed up scheduling and increase placement quality. Regardless of the number of Mapprer, scheduling is fast and provides high-quality placements.

The oracle of EO-GNH evaluates and ranks the meta-heuristic algorithms based on their historical performance. Resulting in the best match between the meta-heuristics and the application.

We implemented asynchronous MapReduce to speed up the execution of meta-heuristics. We compared the proposed parallel model with state-of-the-art distributed meta-heuristics. The result showed that asynchronous MapReduce efficiently provides high-volume non-dominant solutions.

We conducted a simulation-based evaluation. The simulation is used to: (i) dynamically generate requests and vary the number of functions in an SFC (4 to 16) and the function execution time (from 0.5 to 2 seconds); (ii) vary the availability and failure profile of resources using a clock mechanism that aligns resource unavailability with request arrival rate (using Mean-time-to-Failure and Mean-time-to-Recovery metrics). We create a number of possible simulation scenarios to compare EO-GNH with GNH, simple greedy scheduler, random placement, and round robin. On unreliable infrastructure, our results show that with the meta-heuristics, the system is able to have 100% availability of the services, whereas GNH has 95.5%.

In the next chapter (*Chapter 7*), we will evaluate GNH and EO-GNH in a different scenarios. The scenarios are in different environments to test the scheduling quality of the algorithms.

Chapter 7

Performance Evaluation of Adaptability in Intelligent IoT Applications

7.1 Introduction

This chapter aims to comprehensively evaluate the system and scheduling approach introduced in chapters 4, 5, and 6, with an emphasis on their performance and efficiency in computational environments.

We define real-world *intelligent IoT application* as an IoT application that demonstrates genuine intelligence by leveraging machine learning and AI inference to go beyond data collection and storage to enable real-time processing, analysis, and proactive decision-making.

The evaluation employs three unique IoT applications: a machine learning-driven flood prediction model in a *smart city* (*Section 7.2*), federated learning in *agriculture* with a focus on data privacy (*Section 7.3*), and a time-series forecasting model in a *smart factory* for optimising cooling performance (*Section 7.4*).

The objective is to examine the system's operation in a variety of scenarios, analysing its adaptability, capacity for integration, and the efficacy of the proposed scheduling

approach under varying application demands. This chapter represents an important stage in the process of assessing the viability and adaptability of our system within the dynamic landscape of intelligent IoT applications.

This chapter is divided into five sections. The first three sections (following the current section) each examine a distinct intelligent IoT application. For each application, we systematically examine the application components, identify specific requirements and characteristics, describe the experimental setup, and discuss the outcomes. This uniform structure enables a comprehensive, cross-situational analysis of our system, demonstrating its adaptability and efficacy in real-world IoT applications. In *Section 7.5*, we discuss how the solution can be generalised to many IoT applications. Finally, the *Section 7.6* serves as the conclusion for this chapter.

7.2 Flood-Prepared: Cities' Adaptation to Surface Water Flooding

As cities embrace digital technologies, the IoT has become crucial in managing surface water flooding [93]. Through real-time monitoring of drainage systems, IoT can help mitigate flood risks. However, ensuring the reliability of these systems is a challenge, as disruptions could worsen flood situations. Therefore, it is imperative to develop fault-tolerant IoT models to enhance urban resilience against flooding, minimising the effects of potential system failures during such emergencies [94].

In real-time IoT applications such as surface water flood prediction and management, the stakes are high. The unpredictable nature of heavy rainfall and the significant impact of floods necessitate reliable, responsive IoT systems. Traditional fault tolerance approaches, such as redundancy and reactive recovery, often fall short due to limitations in scalability and adaptability. Further, operating within strict QoS criteria and budgetary constraints add layers of complexity [94].

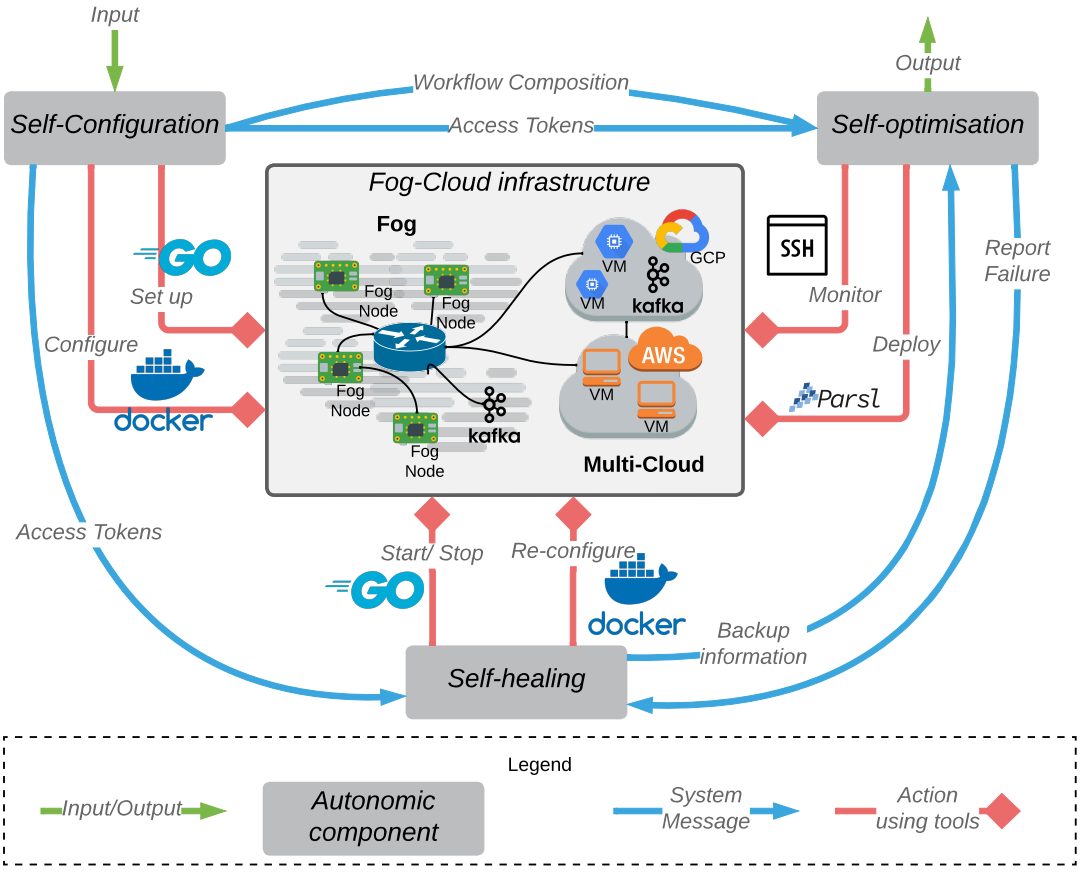


Figure 7.1: System overview of the proposed approach with self-healing, self-configuration, and self-optimisation .

A unique aspect of our methodology is the integration of self-adaptive software mechanisms: self-optimisation and self-healing. These elements permit the system to adjust and recover autonomously upon detection of failures. The self-optimisation component manages IoT application deployments continuously, identifying intolerable errors in the primary resource group. The self-healing component corrects system failures, deactivating failed resources, enabling backup resources, and initiating failed resource recovery 7.1.

We introduce a comprehensive, dynamic IoT fault-tolerance model based on the self-adaptive mechanisms discussed previously. This model combines proactive (self-optimisation with GNH) and reactive (self-healing) strategies for use in real-time scenarios like flood management. The model consists of a controller and a resource pool at its core. The

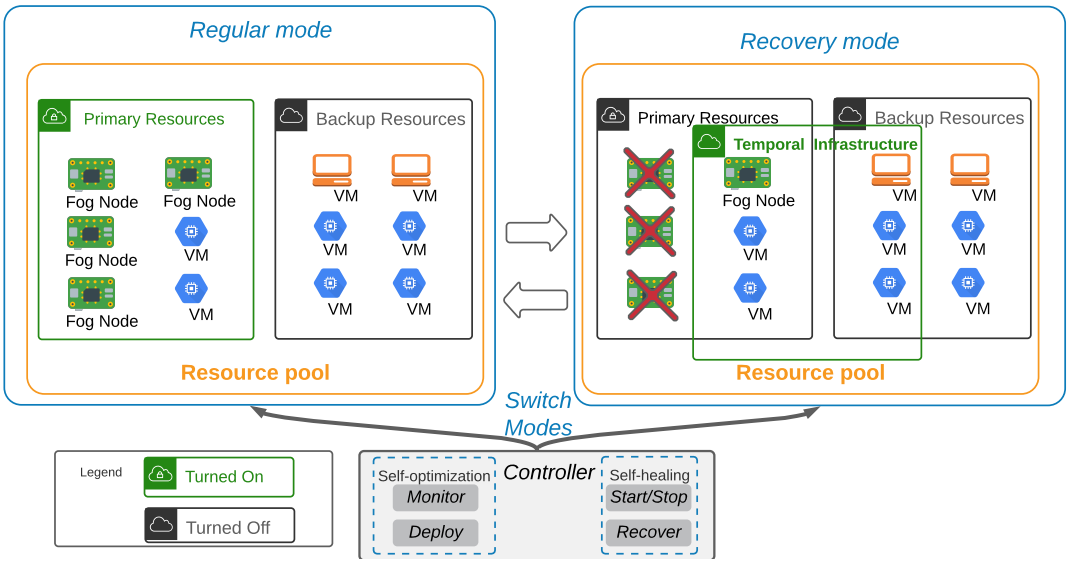


Figure 7.2: Self-healing properties of the system which switches between regular mode and recovery mode .

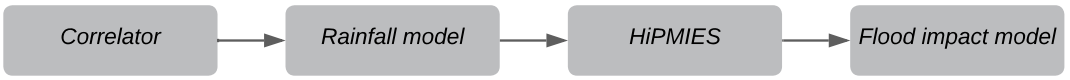


Figure 7.3: Flood-preparation inference workflows

controller manages the resource pool, switching between regular and recovery modes based on system performance and failure detection, as shown in Figure 7.2.

By integrating proactive and reactive fault-tolerance mechanisms, we ensure system reliability and efficient flood management while reducing operational costs. This strategy anticipates and responds immediately to potential failures, thereby avoiding costly downtime. This model's robustness supports its suitability for flood anticipation applications, thereby enhancing urban infrastructure's resilience and effectiveness.

7.2.1 Application workflow

Correlator: the correlator, an essential part of the workflow, reduces data format inconsistencies between models, thereby increasing the pipeline's effectiveness. It con-

sumes data from Apache Kafka, processes it concurrently, and converts it to a unified format for subsequent models using a multi-threaded approach. The ability to handle large volumes of real-time data and easy interoperability are ensured. Figure 7.3 shows the flood-preparation inference workflows.

Rainfall model: A spatiotemporal neural network forecasts rainfall by generating a spatial field of rainfall intensity, enhancing real-time surface water flood predictions. Using physical analytic principles coupled with statistical analytics, this model predicts rainfall distribution across a city ahead of actual events, offering a proactive approach to city-scale flood management [93].

HiPIMS: HiPIMS (High-Performance Integrated hydrodynamic Modelling System) is a 2-dimensional flood model that predicts complex flow dynamics during flooding events. HiPIMS accurately replicates urban overland flows by using depth-averaged shallow water equations and a Godunov-type finite volume scheme [95], while accounting for friction and infiltration rates. A distinguishing feature is the generation of comprehensive surface water flood maps with a 2 metres spatial resolution, which enables proactive measures and strategic planning for potential flood scenarios. Large-scale simulations and real-time forecasting are optimised with GPU for efficiency [93].

Flood impact model: The model is a Convolutional Neural Network (CNN) to estimate vehicle counts at specific locations in order to predict the impact of flooding on urban traffic in real time. It employs techniques for very short-term weather forecasting to extend these predictions to the entire road network. The system employs surface water depths derived from flood models to identify roads that are impassable due to flooding, with a 300 millimetres threshold. It also takes into account speed reductions and congestion caused by water depths below 300 millimetres assuring comprehensive real-time analyses of the effects of flooding [93].

7.2.2 Application Characteristics & Requirements

System integration flexibility: This refers to the adaptability and compatibility required when combining separate systems into a single entity. This includes interoperability, which is the communication capability of system components or self-adaptive components to exchange messages and data that are comprehensible [96].. In addition, portability incorporates the capacity to transfer and reuse software components, such as VM/container, across multiple platforms without compatibility concerns [96]. This flexibility may necessitate infrastructure modifications during application execution in distributed systems in order to meet QoS requirements or manage operational costs in the event of a failure.

Big data constraints: Managing big data requirements necessitates a system that can effectively handle vast data volumes. A particular challenge arises when using Python, due to its garbage collection mechanism. This process frees memory from unused Python objects, but it can cause memory fragmentation, thereby increasing memory usage [97]. Additionally, Python's method of allocating new memory to pools of objects (known as *arenas*) before releasing other memory fragments may gradually increase memory consumption and degrade overall performance.

Applications objectives trade-offs: Applications objectives trade-offs refer to balancing conflicting goals in a system, often seen in mission-critical applications where performance, dependability, and cost need to be balanced. High throughput activities may increase system overhead and reduce reliability, while enhancing dependability can raise costs. This balance becomes more crucial when self-configuration and self-healing components are incorporated into a self-optimisation system, underscoring the need for managing trade-offs to maintain performance, improve reliability, and control costs.

7.2.3 Experimental Setup

This experiment is conducted in a Fog-Cloud environment, utilising Amazon Web Service (AWS) and Google Cloud Platform (GCP). The infrastructure incorporates three GPU nodes (GCP-GPU), all based in the GCP cloud, forming part of an overall system of eight computing nodes divided into primary and backup roles.

HIPIMS, requires a GPU instance with an NVIDIA Tesla P100 to execute CUDA programs. All other functionalities can run on all computing nodes.

The fog infrastructure, located in Cardiff, UK, hosts the controller and Kafka server. It consists of a commodity machine and a Raspberry Pi 4B. The commodity machine has a 6-core CPU and 32 GB of memory. In contrast, the Raspberry Pi 4B, with 4 cores and 4 GB of memory, acts as the controller and runs various virtual instance tasks.

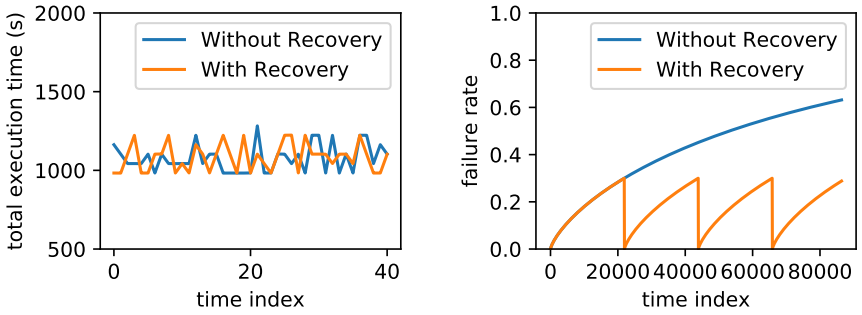
This setup utilises GCP instances, GPU and CPU nodes, situated in the europe-west1-b zone in Brussels, Belgium. It includes 4 GCP-GPU nodes, all of the n1-standard-4 type with 64 vCPUs, 240 GB of memory, and 50GB of storage, 2 of which are primary resources, while the other 2 serves as a backup. Additionally, there are 2 e2-medium GCP-CPU nodes, each equipped with 2 vCPUs and 4 GB of memory. One of these CPU nodes is a primary resource, and the other functions as a backup. The average round trip time (RTT) for a 14B Python object from the fog to either GCP-CPU or GCP-GPU nodes is approximately 99 ms ($\pm$ 24.8).

The AWS resource includes one t2.micro instance, located in London, UK. It has 1 vCPU, 1 GB memory, and a 15 GB disk, serving as a backup. The RTT between AWS and the fog is approximately 93 ms ($\pm$ 4.73).

This experiment makes use of the node performance degradation simulation, which is elaborated in *Chapter 4*, Section 4.4.3. In the parameters for the experiment, we establish a constant value for λ at 86400 seconds, while k is varied from 0.5 to 0.8 across different resources. Specifically, the Raspberry Pi 4B and the AWS instance are assigned a k value of 0.5. The Commodity machine and GCP's e2-medium have a k

of 0.75, and the GPU nodes are assigned the highest value at 0.8. This varying k setup allows for an evaluation of system performance under different resource conditions.

7.2.4 Results



(a) Total execution time comparison (b) Failure rate comparison

Figure 7.4: Results of experiments for the system with and without self-healing The time index is the time of the day in seconds. .

The system features a *self-healing* mechanism enabling it to recover from failures over time, as depicted in Figure 7.4b. The failure rate consistently decreases, demonstrating the continuous effectiveness of the self-healing mechanism. In contrast, a system without self-healing, given the same conditions, would likely experience an increasing failure rate over time, with problems accumulating and compounding.

Metric	AWS	GCP-CPU	GCP-GPU	GCP-GPU	Total
Price per Hour	\$0.0116	\$0.04	\$2.3460	\$2.3460	\$4.7436
Uptime per day (minutes)	18.36	7.54	59.17	60.97	146.04
Bill with self-healing (month)	\$0.11	\$0.15	\$69.40	\$71.52	\$141.18
Bill without self-healing (month)	\$8.35	\$28.80	\$1,689.12	\$1,689.12	\$3,415.39
Decrease backup cost by	98.72%	99.48%	95.89%	95.77%	95.87%

Table 7.1: Compare cost with different adaptive property setups

Typically, the continuity of service demands costly and resource-intensive backup

strategies. However, self-healing systems offer a more resilient and dependable operation by autonomously identifying and resolving potential failures, thereby reducing outage and maintenance costs.

Table 7.1 presents a cost comparison for systems with and without self-healing mechanisms across different cloud computing platforms. The cost metrics are derived from price per hour and uptime per day (in minutes). The stark cost savings achieved through self-healing application implementation are evident across all platforms. For example, on AWS, operating the system with self-healing costs just $0.11 per month, compared to $8.35 per month without it, representing a 98.72% cost reduction.

Figure 7.4a underscores the critical role of the GNH algorithm in managing execution time, both in scenarios with and without self-healing. GNH primarily optimises resource allocation and task scheduling to efficiently utilise resources and minimise overall execution time. It significantly contributes to quicker task completion even in systems lacking self-healing.

The role of GNH becomes even more crucial in self-healing situations, where the system must handle potential failures and initiate recovery procedures. Execution times with self-healing are comparable to those without, testifying to GNH's effectiveness under fault conditions.

Overall, GNH forms the foundation of efficient task execution in both self-healing and non-self-healing scenarios, underscoring the importance of such intelligent, adaptive algorithms in the design of resilient and reliable IoT systems.

Statistical Variation of the Performance

Figure 7.5 shows ten trials of the experiment, which capture the statistical variance in performance. This experiment compares two approaches: one with self-healing (recovery) and the other without. The primary focus of this comparison was on their execution rates. The 'No Recovery Approach' exhibited a minimum execution rate

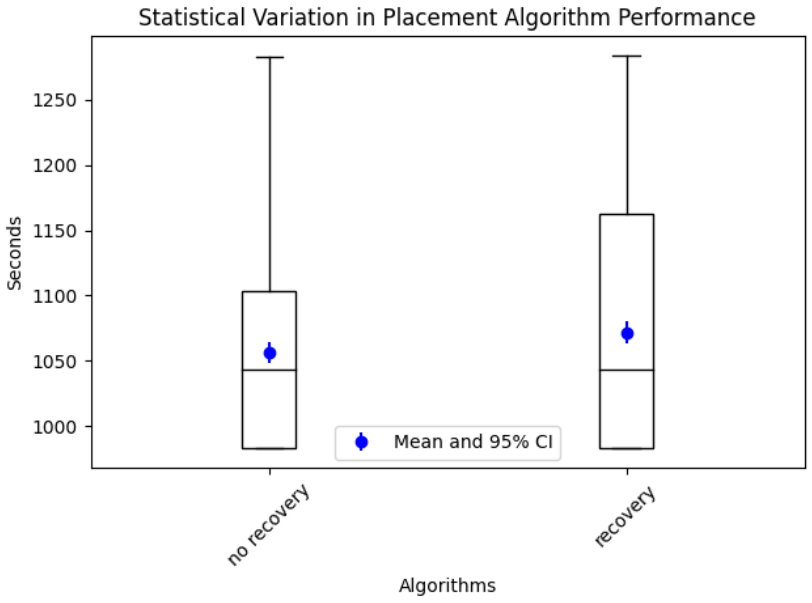


Figure 7.5: For "Self-Healing in Flood Detection": Box plot comparing execution times of self-healing (with recovery) and standard (no recovery) approaches.

of 982.57, a maximum of 1283.13, a mean of 1056.89, and a median of 1042.78. Its first and third quartiles were 983.05 and 1103.07, respectively, with an interquartile range (IQR) of 120.03 and a 95% confidence interval (CI) of ± 7.89. Conversely, the 'Recovery Approach' had similar minimum and median rates but a slightly higher mean execution rate of 1071.80 and a maximum rate of 1283.64. The first and third quartiles for this approach were 983.09 and 1162.69, with a larger IQR of 179.60 and a 95% CI of ± 8.48.

Both approaches displayed similar minimum and median execution rates. However, the recovery approach not only had a marginally higher mean execution rate but also exhibited greater variability, as indicated by its larger IQR. Additionally, the error rates calculated show a marginally lower rate for the recovery approach (16.50%) compared to the no-recovery approach (17.63%). These metrics suggest that the recovery approach, while offering a slightly better average performance, also involves more variability in execution times.

This increased variability in the recovery approach is likely due to the added time for

recovery processes. The recovery approach presumably involves additional steps for self-healing, which can vary in duration based on the complexity of the required recovery. This variation contributes to a broader range of execution times. The type and severity of issues, the system's state, and the efficiency of the recovery process can cause inconsistent invocations or varying time durations for recovery mechanisms. With the recovery process in play, execution times may vary significantly, ranging from highly efficient (with little to no recovery needed) to instances requiring substantial recovery efforts. Additionally, the added complexity of recovery processes likely results in a broader range of behaviours and outcomes, explaining the increased IQR.

The larger IQR in the recovery approach suggests that the inclusion of recovery times contributes to greater variability in execution rates, a trade-off for the added resilience provided by the self-healing capabilities of the system. While it is true that the self-healing approach can add more time to the execution, with an average increase of approximately 14.91 seconds, this additional time could significantly reduce the overall costs associated with GPU nodes. This consideration is crucial when evaluating the effectiveness and efficiency of such systems, particularly in scenarios where balancing performance with resource management and cost-effectiveness is essential.

7.3 Federated Learning in Rural Areas: for Autonomous Weed Detection

Precision agriculture is an innovative approach that leverages data collection and analysis technology, providing significant potential to enhance farming efficiency and food production outcomes [98]. A variety of technologies, such as specialised sensors and satellite-based data, are used to improve farm management. This includes soil and crop knowledge, fertiliser distribution, guidance for farm vehicles and machinery, and product traceability from "farm to fork". A pivotal application within precision agriculture is automated weed control. This approach can potentially boost farm output by

accurately identifying specific weeds, thereby reducing labour costs and potentially decreasing pesticide use. The importance of this application lies in its capacity to reduce agricultural losses, augment productivity, and enhance the environmental sustainability of farming practices.

Addressing privacy and confidentiality concerns in precision agriculture, this research leverages federated learning as an alternative to traditional machine learning methods [99], like those used in autonomous weed spraying. federated learning creates a model using locally sourced data, eliminating the need to transfer data to a central server and thereby enhancing data security. Furthermore, the importance of model training in a fog infrastructure lies in its ability to process data close to its source, enabling efficient, real-time responses, while mitigating some privacy issues — a particularly crucial feature for applications such as precision agriculture.

Rural areas present a unique set of challenges for precision agriculture, mainly due to limited network infrastructure. The implementation of traditional machine learning algorithms can be difficult, which could potentially affect the reliability of the network and availability of services. Thus, this research aims to find a solution that can operate effectively within this constrained network infrastructure, ensuring consistent and effective precision agriculture practices. The goal is to ensure that even in less developed areas, agriculture can benefit from the latest technological advancements without compromising on network reliability or service availability.

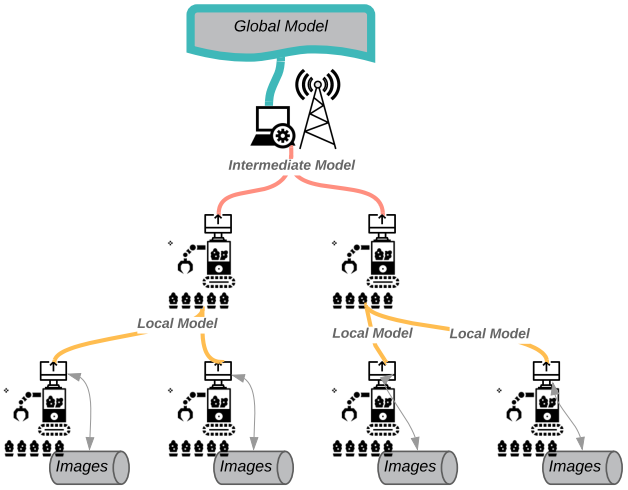


Figure 7.6: Aggregating learned models across robots using federated learning

This research deploys federated learning-equipped mobile robots to address existing challenges. Serving as edge devices, these robots optimise field coverage and data collection via a random walk, contributing to a global model without sharing raw data (Figure 7.6). Moreover, the autonomous robots' trajectories are determined by a truncated random walk approach to ensure efficient field coverage and task accuracy [82]. Importantly, the robot's distance from computing resources affects data communication quality, symbolising network latency in mobile edge devices (Figure 7.7).

7.3.1 Applications Workflows

Image pre-processing: Pre-processing is an essential workflow phase for preparing images for machine learning models. It modifies colour modes, resizes images, transforms image data into an appropriate format, and scales pixel values. The pre-processed images and their corresponding labels (for testing and validation only) are saved for use in subsequent phases of the pipeline. This phase ensures that the data fed into the model is consistent, appropriately scaled, and in a format compatible with the model. Figure 7.8 shows the workflows for federated learning and online training.

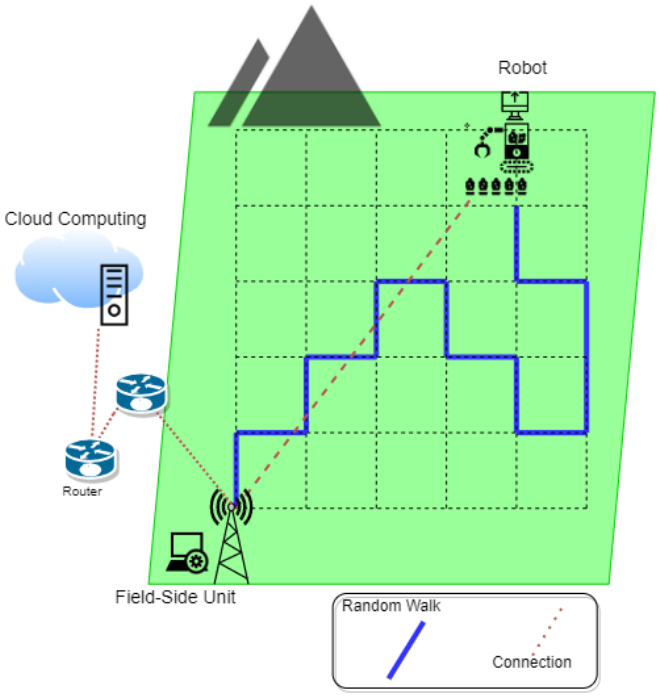


Figure 7.7: Robot performs a random walk, which affects the quality of the connection to a field-side unit. .

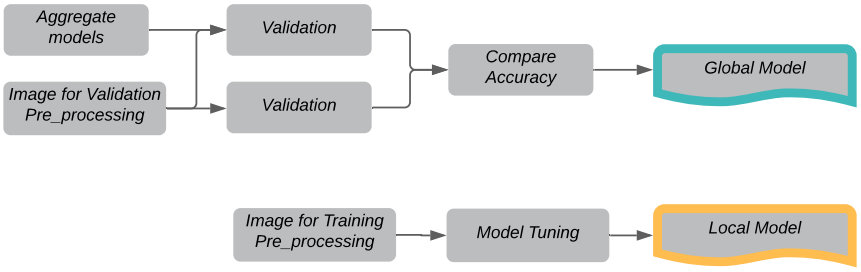


Figure 7.8: Workflows for federated learning online training

Model tuning: In the workflow, model optimisation entails enhancing the performance of an existing machine learning model, specifically a neural network. On a new set of data, the model is fine-tuned by adjusting its weights to better suit the data. The updated model weights are stored separately for future use, facilitating future tasks involving fine-tuning or prediction. This task is essential for enhancing model performance and guaranteeing its adaptability to new or changing data.

Model aggregation: This step in the workflow involves combining the weights of multiple trained models to create a single, aggregate model. It takes in multiple sets of model weights, computes their average, and saves these averaged weights. This results in a new model that captures the collective knowledge of all input models, enhancing the overall prediction performance and robustness. This step is particularly beneficial in distributed learning scenarios, such as federated learning, where models are trained on different datasets or devices and their knowledge is centrally aggregated.

Validation: The model validation step in the workflow is concerned with assessing the performance of the trained machine learning model on unseen data. It loads the model, its weights, and a new set of image data and labels. The model's performance is evaluated on this new data, with metrics such as loss and accuracy calculated. This step is vital for understanding how well the model generalises to new data and for identifying any overfitting or underfitting issues. The validation results, along with the filename of the model's weights, are returned for further analysis or use in the workflow.

Comparing accuracy: This involves comparing the performance of different models to determine the most accurate one. The function takes as input the validation results of two models, comparing their accuracies. If the accuracies are equal, it then compares the loss function values of the models. The function returns the weights file address of the more accurate model (or the one with lower loss in case of equal accuracies). The idea is to retain the best model after each iteration or comparison, leading to continual improvement of the overall model performance over time.

7.3.2 Application Characteristics & Requirements

For AI to be useful in rural regions, a platform must be stable and flexible enough to accommodate changes in system behavior. A list of necessary features for rural AI

applications is outlined below.

Operation during system failure: This requires that applications continue to execute despite intermittent network and computational resource disruptions. It requires fault tolerance, which ensures the system's continued functionality by excluding the assignment of tasks to faulty components. The task scheduler is a crucial component of this process, and it should be designed to minimise the use of resources that are more prone to failure. By doing so, the system will be better equipped to continue functioning and accomplish its intended purpose under challenging circumstances.

Adapt to rural areas conditions: Software systems must be able to operate under the challenging conditions frequently encountered in rural areas. These regions may experience unpredictable internet disruptions at any time, necessitating a reliable mechanism to maintain connectivity with externally hosted services. In addition, regardless of the condition of the local infrastructure, it is essential to ensure that the service quality remains within a predetermined acceptable range. This necessitates that the system be equipped with features that enable it to adapt and maintain performance under less-than-ideal conditions.

Mobility and edge devices: Mobility has a significant impact on application execution performance and the type of data that can be communicated over a wireless network. In particular, the location of mobile edge devices, such as robotics, can influence the data types, sizes, and latency that a network can support. This implies that the design of the application and network must take into consideration the unique challenges posed by mobile devices, such as variable data types, varying data sizes, and potential latency issues.

Managing complexities in federated learning workflows: Different federated learning workflows may have varying computational demands. For instance, a workflow

function with a comparatively longer execution time may require more reliable computing resources. This is to avoid resource reallocation that could result in delays in task completion. Additionally, the outcome of one workflow could be scheduled as the input for another process within federated learning, particularly if the workflows are interdependent. This highlights the need for a system that is capable of efficiently managing these diverse and often complex workflow requirements, ensuring optimal performance and task completion times.

Safeguarding data privacy The edge network must be equipped with the capacity to uphold data privacy effectively. Given that a single application failure can impact the execution speed of other applications negatively, adherence to data privacy requirements during transitions between service instances is essential. Moreover, for enhanced data security and efficiency, it is advantageous to process data within a secure Local Area Network (LAN) rather than on a cloud instance located in a data centre of a cloud provider. Using a LAN network minimises data exposure through the Wide Area Network (WAN) and the internet, further safeguarding data privacy.

7.3.3 Experimental Setup

The experiment's setup initiates with GHN and EO-GNH being evaluated on a Raspberry Pi 4B model referred to as a field-side unit (FSU). FSUs are units situated within the field and, importantly, each robot in this setup acts as a standalone controller. These robots not only execute tasks but also make critical decisions about which functions they should perform and which ones need to be delegated to the FSUs. The robots' location within the agricultural field significantly affects the overall workflow due to varying distances between the mobile robot and FSUs, which are shared resources among the robots.

The experiment simulation generates 3000 application requests to train a local model and 300 to aggregate local models into a global model dynamically. This process is

constrained by parameters contained in Table 7.2. The simulation mimics resource availability and failure profile variability through a clock mechanism, reflecting (i) a failure event with Mean Time to Repair (MTTR) and (ii) regular resource operation with Mean Time to Failure (MTTF).

Variable	Number/Ranges
Field-side units (RPi)	100
Robots (RPi)	10
MTTF	(250 - 500 s)
MTTR	(20 - 100 s)
Random walk steps	(1 - 10 steps)

Table 7.2: Simulation parameters for temperature forecasting experiments

The simulation continues by comparing GHN and EO-GNH to random placement (RP), a simple Greedy algorithm, and Round-Robin load balancing (RR), for a comprehensive performance overview. Application request arrival times and workflow compositions are generated as records, with each workflow deployment constrained by a user-defined deadline, i.e., the expected completion time.

The evaluation phase involves comparing expected and actual completion times to deployed states. If a workflow deployment misses its deadline, it is considered a failure. A failure model parameter for each FSU is randomly selected based on the clock mechanism from Section 4.4.2. Furthermore, the experiment incorporates a random walk and wireless simulation as per Section 4.4.4 to enhance the realism of the simulation environment.

Following each function's placement, a random walk is performed, with the robot moving steps ranging from 1 to 10, representing a shift in the robot's operational environment. This step necessitates regular updates to the failure rate records, assessing the system's resilience and adaptability.

Application requests are distributed uniformly throughout a single day to simulate the

data streams generated by infrastructure controllers (robots). This setup mimics the actual production of data streams.

Following the function placement, a random walk procedure takes place. The robot makes a movement with step lengths ranging from 1 to 10. This randomness emulates the uncertainty and variability of real-world operations, having a significant impact on the experiment's performance.

This random walk is not merely a physical move; it also represents a shift in the operational environment for the robot. Such changes can influence the overall performance and efficiency of the system. Notably, this variability in the robot's movement necessitates frequent updates to the failure rate records, a crucial factor in assessing the system's robustness and adaptability to changing conditions.

7.3.4 Results

In the *Tuning Model* workflow, RP and RR methods have average completion times of 224.98 and 225.5 seconds respectively, with a success rate of around 38%, 37%, respectively and usage of two locations (Figure 7.9). The Greedy method reduces completion time to 214.8 seconds and increases success to 55%, using slightly over one location. However, GNH and EO-GNH variants outperform these, reducing completion times to 159.3 and 152.38-153.35 seconds, respectively, and achieving success rates of 97% and 100%. These methods use about 4.38 to 5.6 locations.

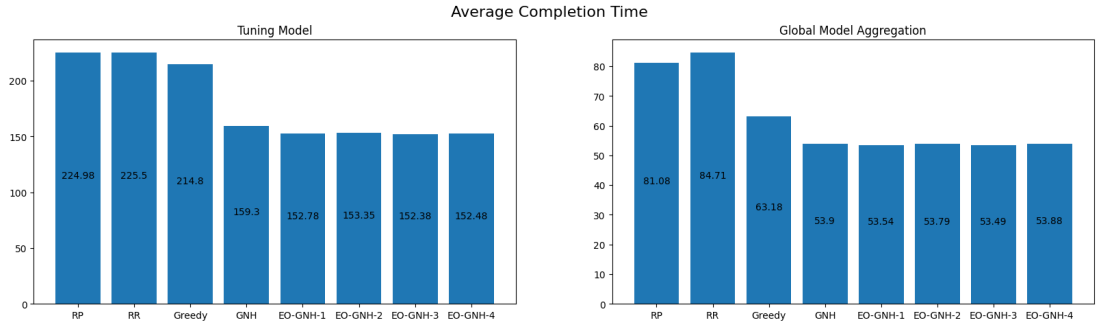


Figure 7.9: Average completion time for federated learning workflows in different algorithms.

In the *Global Model Aggregation* workflow, RP and RR underperform, with completion times around 81.08, and 84.71 seconds respectively and success rates below 36%, using close to 5 locations. The Greedy method improves on this with a 63.18-second completion time and 77% success rate, using just over one location. The GNH and EO-GNH variants excel with completion times around 54 seconds and a 100% success rate, using around 4.43 to 4.81 locations.

In terms of reliability, EO-GNH and GNH demonstrate superior performance, successfully completing most of their requests ahead of schedule, as demonstrated in Figure 7.10. The simple Greedy method achieves a 77% success rate but is outperformed by EO-GNH and GNH. Notably, RR and RP result in the highest number of requests completed after the deadline, highlighting their lower reliability.

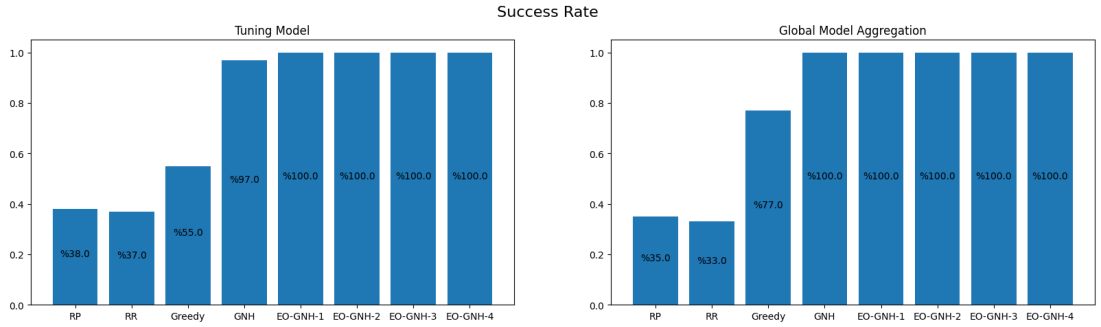


Figure 7.10: Successful rate for federated learning workflows in different algorithms .

EO-GNH and GNH distinguish themselves by strategically selecting multiple FSUs to handle requests, ensuring that a failure in one does not halt the operation. This strategy of redundancy results in a lower failure rate and higher reliability. Notably, all global model aggregation requests are completed earlier than expected, and only 3% of GNH's Tuning Models requests exceed their expected completion times.

Resource utilisation plays a vital role in cost and efficiency, as seen in Figure 7.11. GNH minimises resource usage by deploying an average of 4.81 FSUs. In comparison, RP, due to occasional reuse of resources, averages at 4.99 units. The Greedy algorithm uses 1.51 FSUs on average, which is the least costly manner to run a workflow.

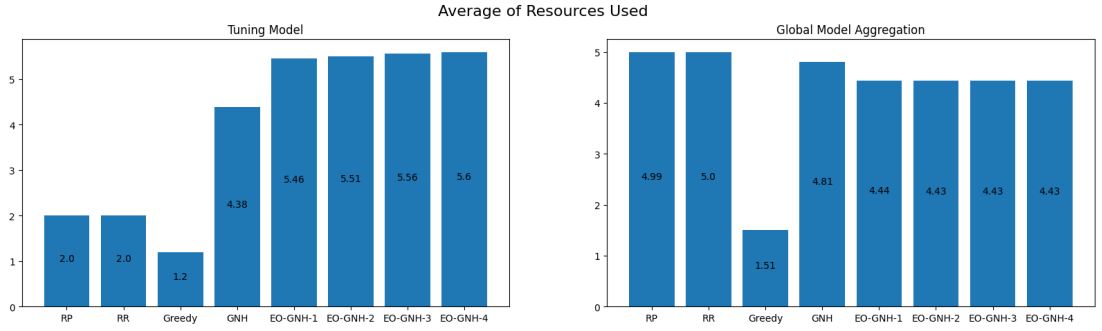


Figure 7.11: Average cost based on the number of locations utilised for federated learning workflows in different algorithms .

As depicted in Figure 7.9, the EO-GNHs and GNH algorithms optimise completion time effectively. GNH significantly outpaces the Greedy method in both tuning model execution (by 55.5 seconds) and global model aggregation (by 9.28 seconds). Compared to RR and RP, EO-GNHs and GNH can save up to 73.12 seconds for model tuning and 29 seconds for model aggregation. This represents a significant savings in terms of time. This results in a considerable reduction in time.

Increasing the number of mappers employed by the EO-GNH algorithm enhances the service availability rate. Specifically, using three mappers (EO-GNH-3) presents an optimal balance between low decision-making costs and high availability, resulting in zero failures. For Global Model Aggregation, EO-GNH-2 performs slightly faster than

EO-GNH-3, offering potential time and cost savings.

Statistical Variation of the Performance

Tuning Model Figure 7.12 summarises the performance of various optimisation algorithms in *Model Tuning*, focusing on their statistical behaviour in a minimisation problem. The execution times of the GNH algorithm vary between 134.02s and 310.79s. Its average (mean) performance is 159.30s, with a median of 154.05s. This algorithm shows a considerable (IQR) of 26.87s, indicating a wider dispersion of values. In contrast, the GDE3 algorithm exhibits a slightly narrower range, with a mean of 157.78s and a median of 152.75s. Both HYPE and IBEA demonstrate tighter clusters in performance, as reflected in their lower IQRs of 24.81s and 23.99s, respectively. The MOCell and NSGAII algorithms show even more concentrated execution times, with IQRs of 22.99s and 22.32s, respectively. These figures suggest that these algorithms have more consistent performance, with most of their execution times clustering around the mean of approximately 154 seconds.

OMOPSO and SMPSO maintain this trend of narrow IQRs at 21.48s and 21.36s, respectively, with mean values just above 153s, signifying consistent performance. The EO GNH series, iterating from 1 to 4, consistently shows a median performance around 149s and decreasing IQRs, which suggests a refinement in algorithmic efficiency with each successive version. The 95% confidence intervals for these algorithms are relatively tight, all below 0.77s, indicating precise estimates of the mean execution times and underscoring the reliability of these algorithms in solving the considered minimisation task.

Figure 7.13 illustrates the performance disparities among three placement algorithms during Model Tuning, specifically in terms of their execution time statistics. In this context, Model Tuning refers to the process of adjusting the parameters or configurations of these algorithms to optimise their performance. Random Placement, one of these algorithms, demonstrates the broadest range of execution times, from approxim-

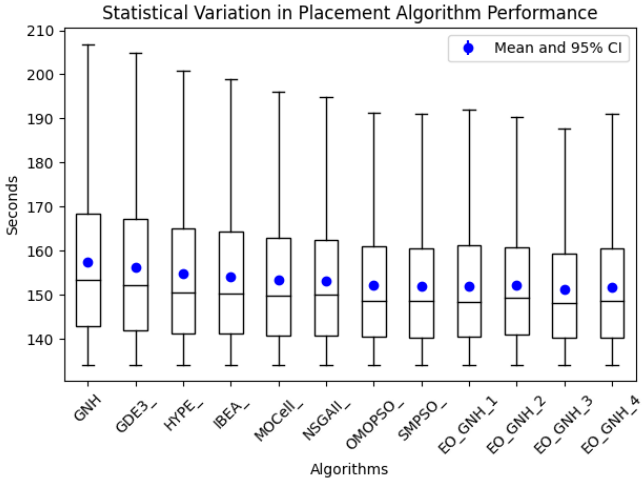


Figure 7.12: For "Model Tuning": Box plot showing execution times for algorithms. Median and mean (blue dot with 95% CI bars) indicated. EO-GNH series highlights efficiency gains with added mappers. .

ately 134.05s to 491.30s, with an average time of 224.98s and a median of 208.16s. This broad range indicates a considerable spread in data, as evidenced by an IQR of 86.61s, the largest among the three algorithms. The Round Robin algorithm shows a slightly less varied range, with execution times from around 134.16s to 476.14s, a mean slightly higher than Random Placement at 225.50s, and a median of 212.41s. Its IQR of 85.48s is marginally less than that of Random Placement, suggesting a somewhat tighter clustering of execution times.

In the context of Model Tuning, the Greedy algorithm, while having a wider range, demonstrates improved performance with execution times ranging from 134.04s to 440.46s. It has a lower average time of 214.80s, and a median of 199.13s, indicating more central tendencies compared to the other two algorithms within the Model Tuning workflow. The Greedy algorithm's IQR of 76.01s is the smallest among the three, suggesting a tighter concentration of values. The 95% confidence intervals for these algorithms are relatively large, all above 2.30s, pointing to less precision in the mean estimation with broader data dispersion. However, the Greedy algorithm stands out with slightly more precision in its mean estimation, as evidenced by its 95% confidence interval of 2.33s. While this interval is somewhat higher, in the context of a lower

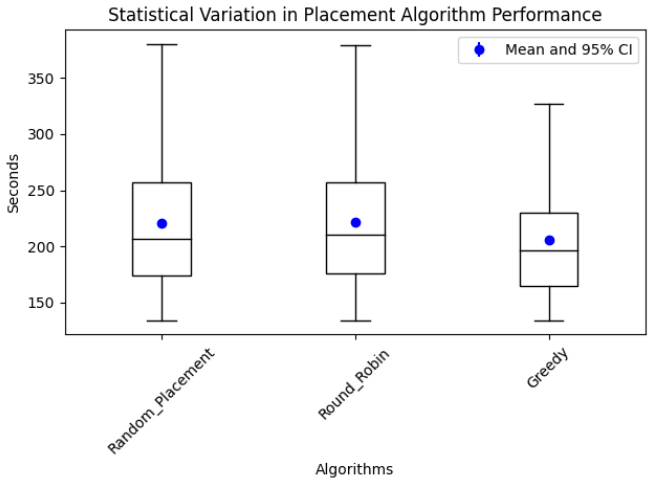


Figure 7.13: For "Model Tuning": Box plot comparing execution times of Greedy, Random Placement and Round Robin algorithms. Median and mean (with 95% CI bars) are highlighted. Shows efficiency comparisons..

mean, it indicates a more efficient performance in the minimisation problem tackled through Model Tuning.

Models Aggregation Figure 7.14 reflects the tightly clustered performance metrics of several optimisation algorithms within the models aggregation workflow, each demonstrating efficiency in a minimisation context. The GNH algorithm, as part of this workflow, shows a moderate range of execution times with a minimum of around 49.09s and a maximum of 65.64s, while maintaining an average of 53.90s and a median close to 53.14s. Similarly, the GDE3 algorithm follows closely, exhibiting a slightly narrower range and a mean of 53.56s, indicating consistent execution times within the same workflow. The HYPE algorithm extends the maximum execution time slightly more but maintains a competitive average of 53.87s, further illustrating the consistency and efficiency of these algorithms in the context of the models aggregation workflow.

Within the models aggregations workflow, the IBEA algorithm presents a wider range yet manages to keep its mean at 53.77s with a median marginally lower, suggesting balanced performance. MOCell and NSGAII report similar behaviours with tight IQRs and means just above 53.50s, indicative of their stable nature in solving the problem.

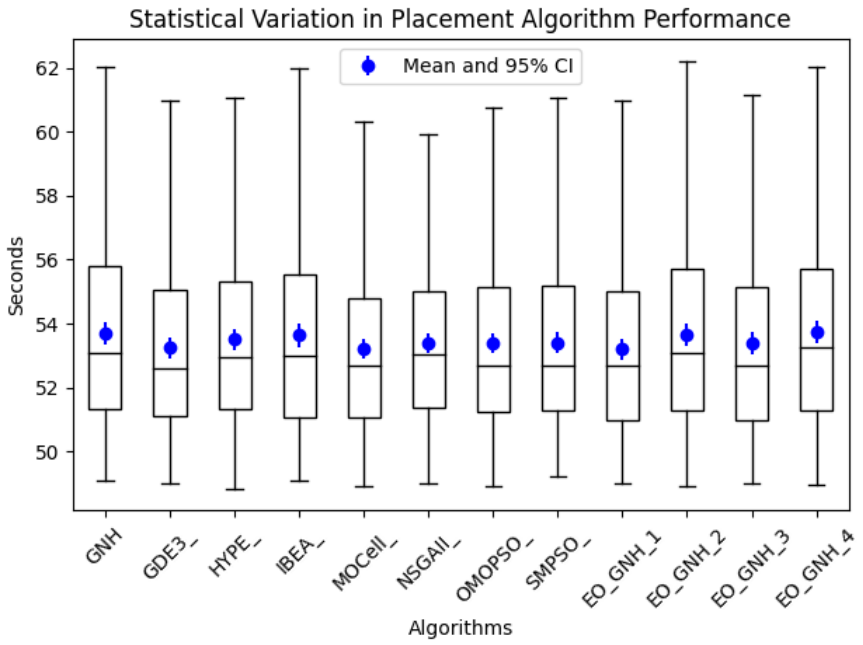


Figure 7.14: For "Models Aggregation": Box plot showing execution times for algorithms. Median and mean values (blue dot with 95% CI bars) are also indicated. EO-GNH series highlights efficiency gains with added mappers..

OMOPSO expands the range further but still delivers a mean of 53.68s, while SMPSO maintains this pattern with a mean of 53.72s, demonstrating the efficiency and consistency of these algorithms in the context of the models aggregations workflow.

The EO GNH with different setups, executing models aggregations workflow, encompassing iterations from 1 to 4, displays a consistent median around 53.00s and slightly increasing IQRs. Despite the increasing IQRs, this still indicates a refinement in performance with each successive version. The 95% confidence intervals for all algorithms are modest, varying from around 0.36s to 0.38s. This provides confidence in the stability of the mean execution times and underscores the algorithms' effectiveness in consistently achieving near-optimal solutions as part of the models aggregations workflow.

Figure 7.15 provides a detailed analysis of the performance of three distinct placement algorithms within the models aggregations workflow, displaying varied execution time ranges and central tendencies. The Random Placement algorithm, as part of this

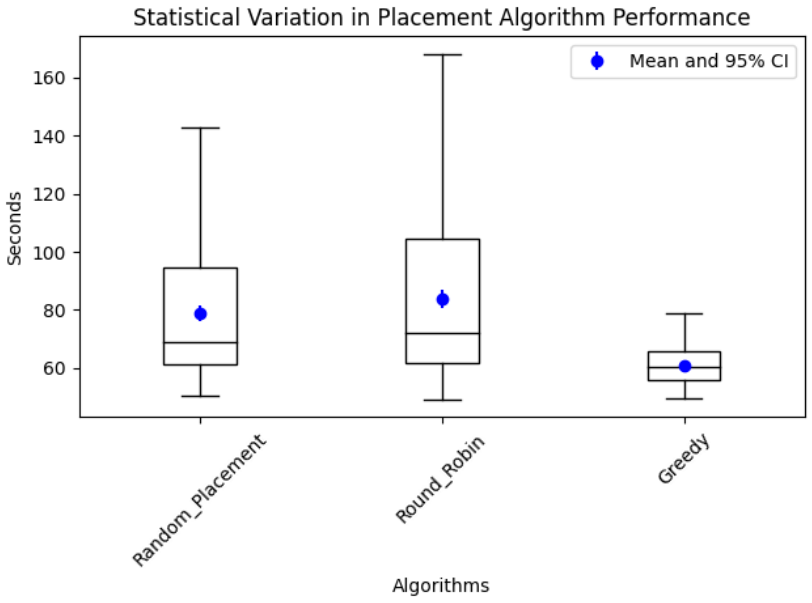


Figure 7.15: For "Models Aggregation": Box plot comparing execution times of Random Placement and Round Robin algorithms. Median and mean (with 95% CI bars) are highlighted. Shows efficiency comparisons..

workflow, exhibits a wide range of execution times, with a minimum of approximately 50.43s and a maximum of 201.10s. This results in a mean time of 81.08s and a median significantly lower at 69.32s. The variation between the mean and median, coupled with a substantial IQR of 34.62s, points to a diverse distribution of execution times within the context of the models aggregation workflow.

The Round Robin algorithm running the models aggregation workflow demonstrates an expanded range from around 49.08s to 182.78s, with a mean of 84.71s and a median of 73.21s, higher than Random Placement. The IQR of 42.92s is the largest among the three, indicating a greater spread of execution times and less consistent performance compared to the other algorithms.

In contrast, the Greedy approach running models aggreagation workflow presents a notably tighter performance range, with a minimum and maximum between 49.33s and 145.23s. It achieves a mean of 63.18 and a median of 61.22s, both markedly lower than those of the other two algorithms. The IQR of 10.43s is significantly smaller,

suggesting a more concentrated and consistent set of execution times. The CI of 1.35s for the Greedy algorithm is considerably smaller than those of Random Placement and Round Robin, which have CIs of 3.01s and 3.26s, respectively. This indicates a higher precision in the Greedy algorithm's performance, making it a more efficient choice for solving the minimisation problem at hand.

7.4 Intelligent Cooling System: Cooling Fish Processing Facility

The complexities of food production, coupled with stringent regulations set by food authorities, create challenges for the food industry, particularly in maintaining safety and quality standards. A common practise within this sector is the use of climate-controlled storage, such as refrigerated and frozen rooms, to preserve regular food batches [100].

However, with the advancement of technology, there is an opportunity to significantly improve efficiency and quality assurance in these facilities. Specifically, an IoT temperature monitoring system (Figure 7.16) can be deployed. This system offers an automated solution to monitor and manage product quality, preserve required storage temperatures, and ensure regulatory compliance through real-time monitoring of facility temperatures.

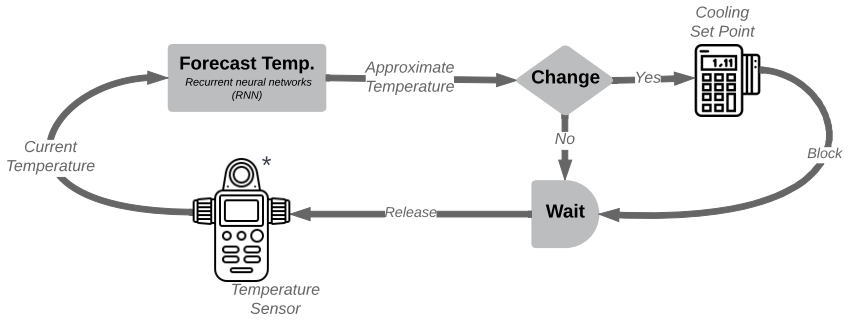


Figure 7.16: Temperature control that forecasts temperatures. Sensors are the source of the feedback loop, from which it gathers data and deduces a predicted temperature. The set point is updated based on the prediction. .

In the pursuit of operational efficiency, AI and Recurrent Neural Networks (RNNs) are important. RNNs, known for their capability in handling time series forecasting, provide invaluable insights by predicting future temperatures based on collected sensor data, enabling the system to adapt the temperature setpoint accordingly. This reduces energy waste, contributing to substantial cost savings.

With this scenario in mind, the research will focus on a specific application case study of an efficient, smart energy cluster in a food processing facility. The facility stands to make significant energy savings through the incorporation of an RNN designed for energy conservation. This RNN predicts room temperature and energy consumption, auto-adjusts the temperature setpoint, and, as a result, produces accurate energy usage and temperature forecasts.

The implementation of the RNN within the facility is strategically distributed into various service functions, as illustrated in Figure 7.17. This distribution represents an innovative and efficient workflow management. The RNN architecture incorporates a pre-processing layer responsible for preparing and processing the incoming data. Following this is a hidden layer that performs complex computations and carries the main predictive responsibility. The output layers deliver the predicted values for energy usage and temperature. This structured workflow demonstrates a robust and effective

utilisation of advanced AI technology in the facility.

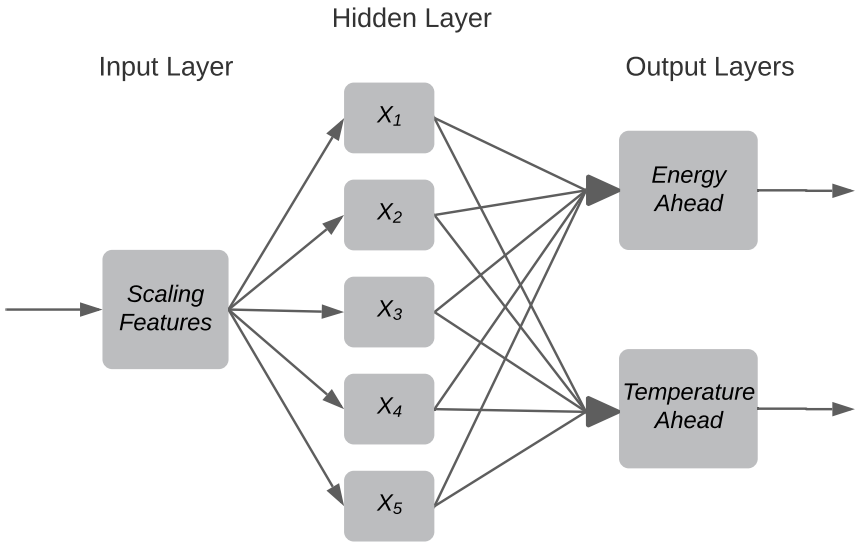


Figure 7.17: Workflow of neural network for energy-saving applications

7.4.1 Applications Workflows

Scaling features: This phase is a important component of the pre-processing that prepares the input data for the RNN. It is designed to accept multiple inputs, such as the current readings of the chamber's set points, power, and capacity, as well as the current season. The procedure entails transforming these separate inputs into a standard scale that is compatible with the neural network. Each input variable receives a specific formula that maps its value to a predetermined scale. This step guarantees that all input data is normalised and in a format that the RNN can effectively process, thereby enhancing the model's capacity to learn and make precise predictions.

Neurons X_1 to X_5 : These are neurons in the RNN layer of neural networks, positioned in the middle, where the function applies weights to inputs and guides them through an activation function as the output. Hyperbolic Tangent (Tanh) activation is

used by the neurons. The Scaling Features function output is multiplied by the weight of the neurons, and the sum of the multiplied values is sent to the Tanh activations funciton. Each neurons output is result of the following calculation $Tanh(\sum_{i=1}^{n} I_i \cdot w_i)$ where i is the index of the inputs of the neurons, and n is the input size.

Energy ahead and temperature Ahead : Both operate provide energy and temperature's forecasts. Each function has layers which are output layer, unscaling layer, and bounding layer, connected in the order listed. The output layer is the sum of the X_j layers outputs multiplied by the output layer's weights, calculated by the following formula: $\sum_{j=1}^{5} O_j \cdot w_j$ where O_j is the outputs of neuron X_j. The output Layer results is scaled, the unscaling layer produce the output to the original units, kWh for energy and celsius degrees for temperature. The unscaling layer is calculation based on statistics on the outputs layer's results, such as minimum and maximum values. Finally, the Bounding layer ensures that the output is within valid value boundaries.

7.4.2 Application Characteristics & Requirements

High availability in forecasting: Due to its inherent structure, a distributed approach can offer high availability in temperature controller services, which are beneficial in limiting single points of failure. Nonetheless, managing this distribution effectively is critical to prevent infrastructure congestion. Adding replication of tasks could serve as an additional safety measure. However, there needs to be a balance between maintaining enough replicas for high availability and reducing congestion in the system. Therefore, the strategic management of task replication becomes critical in maintaining high availability while mitigating system congestion.

Concurrency: Employing multiple fog nodes can effectively speed up RNN inference through concurrent execution. By allowing tasks to run in parallel, we can avoid

the delays associated with waiting for independent tasks to finish before others can begin. Without parallel execution, any delay in one task would hold up the progress of others, even if they are not interdependent. Therefore, enabling concurrency in the distributed system is of significant importance in this context.

7.4.3 Experimental Setup

In this experimental setup, GNH and EO-GNH algorithms are assessed within a fog infrastructure, designated as Fog Nodes (FNs), modelled by Raspberry Pi 4B units. The simulation incorporates one thousand requests for executing RNN inference. Two distinct configurations are deployed, involving 100 and 1000 locations respectively. This process is constrained by the parameters in Table 7.3.

Variable	**Number/Ranges**
Fog Nodes (FN)	(100, 1000)
Controller	1
MTTF	(250 - 500 s)
MTTR	(20 - 100 s)

Table 7.3: Simulation parameters

7.4.4 Results

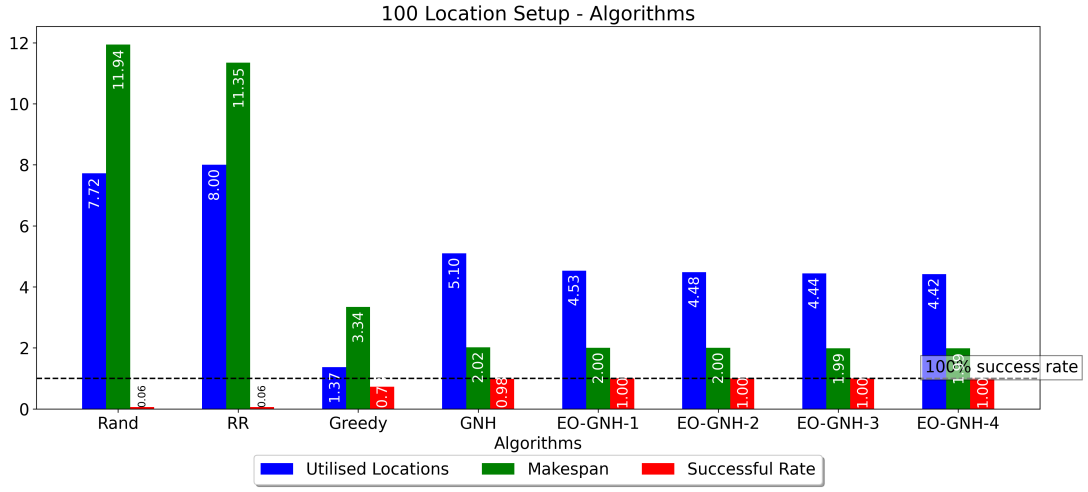


Figure 7.18: Performance comparison of various algorithms in a 100 location setup for the Distributed RNN application.

Figure 7.18 shows the results of the 100 location setup, highlighting the performance of various algorithms in this context.

Rand and RR utilised a significant number of locations, approximately 7.72 and 8, respectively, but they also had the longest completion times, approximately 11.94 and 11.35 seconds. Their success rates were remarkably low at just 6% for both.

On the other hand, the Greedy algorithm used fewer locations (on average around 1.37), and also achieved a much quicker completion time of 3.34 seconds, with a significantly higher success rate of 73%. The GNH algorithm managed to reduce the completion time further to 2.02 seconds, using around 5.1 locations, and achieved a high success rate of 98%.

All versions of the EO-GNH algorithm, utilising mappers from 1 to 4, demonstrated the best performance. They achieved the highest success rate of 100% and completion times of approximately 2 seconds. The number of locations utilised decreased marginally from EO-GNH 1 to EO-GNH 4, averaging around 4.5 locations.

Performance of different mappers within the EO-GNH framework in a 100 location setup for the distributed RNN application. Each mapper runs a single meta-heuristic and utilises a greedy approach for the Reducer.

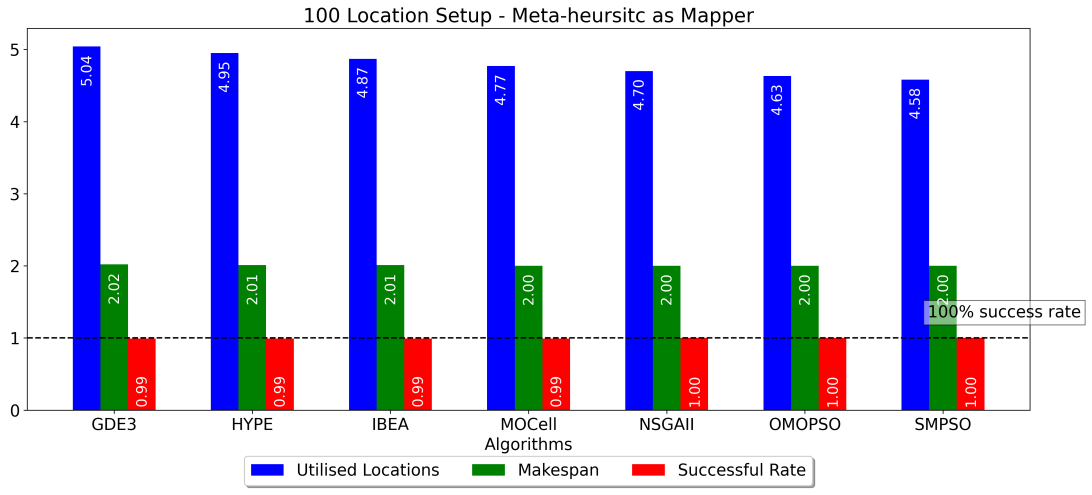


Figure 7.19: Performance of different mappers within the EO-GNH framework in a 100 location setup for the Distributed RNN application.

Figure 7.19 shows the results of the 100 location setup with the implementation of different mappers within the EO-GNH framework, each running single meta-heuristic (Mapper) and utilising greedy approaches for the Reducer. GDE3 used approximately 5.04 locations, resulting in a completion time of 2.02 seconds, and achieved a high success rate of 99%. HYPE slightly reduced the locations to 4.95 and the completion time to 2.01 seconds, also with a 99% success rate. IBEA improved the efficiency further, utilising 4.87 locations and maintaining a completion time of 2.01 seconds and a success rate of 99%. MOCell was able to reduce the number of locations to 4.77 and the completion time to 2.0 seconds. It too achieved a 99% success rate.

NSGAII, OMOPSO, and SMPSO all reached the optimum success rate of 100%. NSGAII utilised approximately 4.7 locations and a completion time of 2.0 seconds. OMOPSO reduced the locations slightly to 4.63, while SMPSO used even fewer at 4.58 locations. Both maintained a completion time of 2.0 seconds.

Summarising the results from the 100 location setup, all the mappers within the EO-GNH framework displayed high success rates and efficient use of locations. However, NSGAII, OMOPSO, and SMPSO achieved the optimum success rate of 100% with reduced locations and minimal completion times, demonstrating superior performance under these conditions.

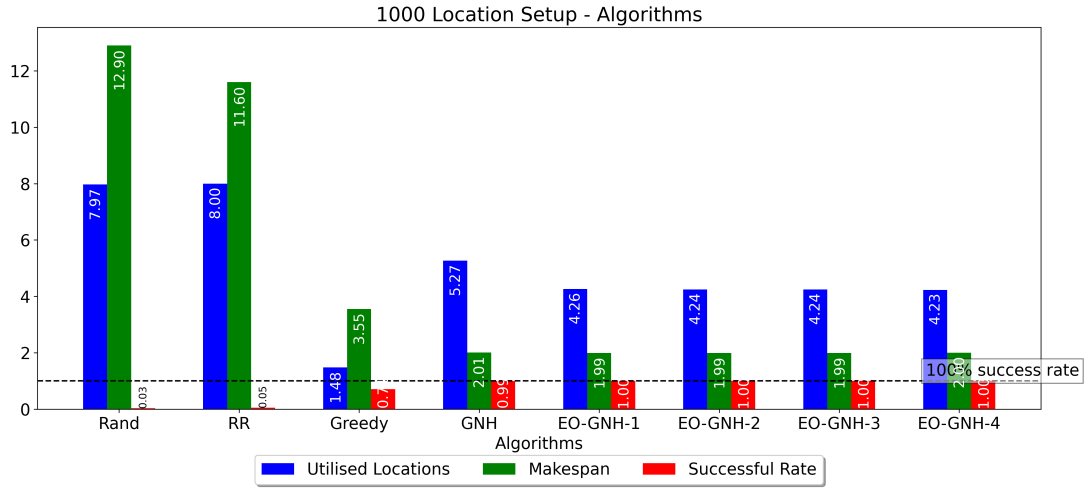


Figure 7.20: Comparison of algorithmic approaches in a 1000 location setup for the Distributed RNN application RNN application.

Figure 7.20 gives the results from a 1000 location setup using a variety of algorithmic approaches.

The Rand algorithm acquired approximately 7.97 locations, with a high completion time of 12.9 seconds and a notably low success rate of 3%. The RR algorithm performed slightly better, utilising 8 locations with a shorter completion time of 11.6 seconds and achieving a slightly higher success rate of 5%.

The Greedy algorithm made a significant leap in performance. It only required 1.48 locations, drastically reduced the completion time to 3.55 seconds and achieved a success rate of 71%. GNH showed further improvement, using approximately 5.27 locations but dramatically reducing the completion time to 2.01 seconds and achieving an impressively high success rate of 99%.

The four configurations of EO-GNH (1 to 4) demonstrated superior performance. EO-GNH-1 used 4.26 locations and had a completion time of 1.99 seconds, with a perfect success rate of 100%. EO-GNH-2 and EO-GNH-3 demonstrated identical performance, both utilising 4.24 locations with a completion time of 1.99 seconds and maintaining the 100% success rate. EO-GNH-4 showed very similar performance, with 4.23 locations used and a slightly increased completion time of 2.0 seconds but also with a perfect success rate.

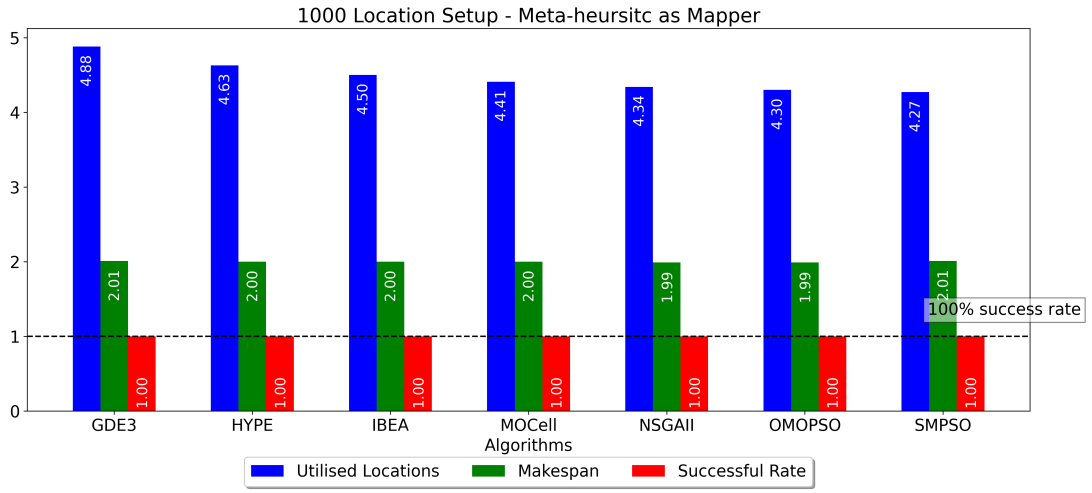


Figure 7.21: Results of the 1000 location setup under the EO-GNH framework for the Distributed RNN application.

Figure 7.21 presents the results of the 1000 location setup under the EO-GNH framework with one mapper running single meta-heuristics and reducers employing greedy approaches. Each meta-heuristic shows differences in obtained locations and completion times, but they all exhibit a 100% application success rate.

The GDE3 meta-heuristic obtained 4.88 locations with a completion time of 2.01 seconds. This is the highest location utilisation among the mappers, but it does not reflect a slower performance as the completion time remains competitive.

The HYPE meta-heuristic decreased the obtained locations to 4.63, while slightly reducing the completion time to 2.0 seconds, aligning with the ideal trend of lower location

utilisation and faster completion. The IBEA meta-heuristic further reduced location utilisation to 4.5 with an identical completion time to HYPE at 2.0 seconds, indicating a comparable but slightly more efficient performance.

MOCell used 4.41 locations, maintaining the same completion time of 2.0 seconds, suggesting minor improvement in location efficiency while maintaining speed. NSGAII showed continued improvement, reducing locations to 4.34 with a slightly faster completion time of 1.99 seconds, marking the first mapper to break the 2-second barrier.

The OMOPSO meta-heuristic followed closely, using 4.3 locations and matching NSGAII's completion time of 1.99 seconds. Finally, the SMPSO meta-heuristic utilised the fewest locations at 4.27, but slightly increased the completion time to 2.01 seconds.

Summarising the results from the 1000 location setup, all meta-heuristics delivered a 100% application success rate. However, variations in the obtained locations and completion times provide important metrics for comparative analysis. While GDE3 utilised the most locations, SMPSO was the most efficient, even though it showed a slight increase in completion time. This data provides a useful comparison of the trade-off between location utilisation and time efficiency among different meta-heuristics within the EO-GNH framework.

In conclusion, both the 100 and 1000 location setups indicate that the EO-GNH algorithms are consistently reliable with an optimal success rate of 100%. These algorithms also exhibit rapid completion times, roughly around 2.0 units, which outperforms other methods such as Random Placement and Round Robin. Nonetheless, a slight reduction in the number of obtained locations is observed as the setup size expands, particularly in the Mapper cases.

The GNH method presents a well-rounded performance, achieving a high success rate (98% - 99%) coupled with modest completion times (2.02 - 2.01 seconds), and a commendable count of obtained locations (5.1 - 5.27).

A specific note is to be made for the 1000 location setup, wherein the Mapper segment utilises various meta-heuristics. Each of these methods reaches an exceptional success rate of 100%. GDE3 is prominent among these for securing the highest number of locations at 4.88. Meanwhile, the completion times remain relatively uniform for these approaches, approximately around the 2.0 seconds.

The data thus highlights a balance in algorithm performance. While certain methods excel in specific measures such as obtained locations or success rate, a comprehensive performance across all measures is demonstrated by others like the EO-GNH and selected meta-heuristics.

Statistical Variation of the Performance

This section studies the performance of the RNN-based refrigeration cooling system within an RNN-based workflow. The execution times for various placement algorithms, as illustrated in Figure 7.22, show notable uniformity in their performance. This is evidenced by patterns indicating tight clustering of data and high precision in measurements. Within the RNN-based workflow, the Greedy algorithm exhibits a broader spectrum of execution times, indicated by its IQR being approximately 1.095s and a 95% CI pointing to a moderate level of variability, with average execution times around 3.766s. Conversely, algorithms such as GNH, GDE3, HYPE, IBEA, MOCell, NSGAII, OMOPSO, and SMPSO demonstrate more narrow IQRs, suggesting a more compact distribution of their execution times, with average values generally hovering around 2.00s. This pattern reflects stable performance across these algorithms within the RNN-based workflow, underscored by their low Standard Error of the Mean (SEM) values, ranging approximately from 0.015s to 0.020s, indicating high precision in their average execution times.

Upon evaluating the EO-GNH algorithm within the RNN-based workflow with varying numbers of mappers, there is an observed gradual reduction in average execution time as more mappers are included, hinting at enhanced efficiency. This trend

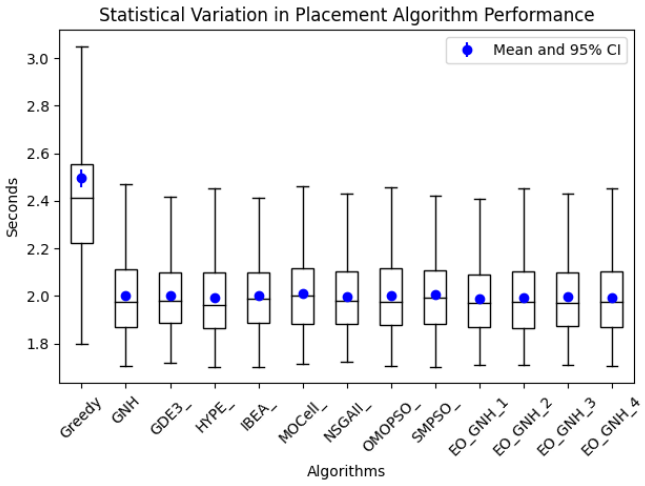


Figure 7.22: For "Energy Forecasting": Box plot showing execution times for algorithms. Median and mean (blue dot with 95% CI bars) indicated. EO-GNH series highlights efficiency gains with added mappers..

is depicted across four different EO-GNH configurations, with mean execution times slightly declining from EO-GNH-1 to EO-GNH-4. All configurations maintain average times under 2.0s, exhibiting close 95% CIs and small SEMs. These findings underscore the algorithm's consistent ability to attain near-optimal solutions within the RNN-based workflow, emphasising its effectiveness in addressing the minimisation challenge presented.

Figure 7.23 shows that within the RNN-based workflow, the Random Placement algorithm displays a broad spectrum of execution times, ranging from 1.84s to 37.75s, with an average time of around 12.14s and a median slightly lower at 11.74s. This range, evidenced by an IQR of 8.56s, indicates a wide variability in its performance. In comparison, the Round Robin algorithm, also part of the RNN-based workflow and showing a diverse range of execution times, has a slightly better average of around 11.78s and a median of 11.38s. Its IQR is marginally narrower at 8.72s, coupled with a 95% CI of 0.38s, suggesting a more consistent grouping of sample means than the Random Placement. This points towards a more stable yet still varied performance profile within the RNN-based workflow.

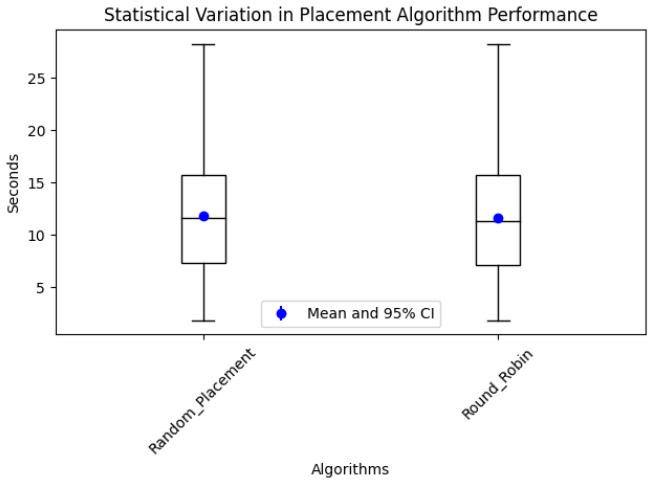


Figure 7.23: For "Energy Forecasting": Box plot comparing execution times of Random Placement and Round Robin algorithms. Median and mean (with 95% CI bars) are also highlighted. Figure also shows efficiency comparisons between algorithms..

7.5 Discussion

We have explored the robustness, adaptability, and generalisability of our proposed solutions, GNH and EO-GNH, across diverse IoT applications, with a specific emphasis on effectiveness and efficiency in terms of execution time, cost, and risk management. These solutions were assessed across a wide range of scenarios, each with varying degrees of complexity, data flow, and resource constraints, including agriculture, factory operations, and smart cities.

We prioritised developing a system characterised by seamless integration, efficient data management, application diversity, and robust system failure resilience. Seamless integration was ensured through self-optimisation components with self-healing and self-configuration capabilities. Efficient data management was achieved via federated learning within the fog infrastructure, safeguarding data privacy. Application diversity was confirmed by testing our solution in three different scenarios: smart city, precision agriculture, and smart factory. We defined 'success rate' as the application's ability to complete execution within the expected time without any failure or delay,

which underscores our approach's adaptability across various conditions. Despite the infrastructure's state, the system shows failure resilience by maintaining a high success rate for deploying the application, most of the time achieving 100% service availability.

In assessing performance, we employed a dual-metric approach focusing on solution quality and convergence speed. For solution quality, we focused on execution time, cost, and the risk of not completing on time. In terms of convergence speed, we found that in both EO-GNH and GNH, the convergence time had a minimal impact on the completion time.

Our system, operated by GNH and EO-GNH, demonstrated superior results when compared to baseline approaches such as Round Robin and Random Placement, as well as a basic greedy approach. This further reinforces the potential of our solution for broad application across various IoT scenarios.

In the field of optimisation, We discovered the need to transition away from a one-size-fits-all strategy, leading us to propose an approach that provides algorithms tailored to specific applications. While GNH was effective in certain contexts, it did not consistently yield optimal results across all applications. This was particularly evident in the federated learning Workflow, where GNH showed varying success rates for model tuning (96%) and model aggregation (100%). In cases where GNH fell short, EO-GNH showcased its adaptability by selecting the most appropriate meta-heuristics to achieve a 100% success rate, resulting in improved execution time.

EO-GNH excels in its adaptability, leveraging the characteristics of the workflow and infrastructure to compose an optimal strategy for the task at hand. This resulted in a significant performance improvement in our model tuning workflow, achieving a success rate of 100%.

In both agricultural and factory settings, our solution maintained a 100% deployment success rate, often completing execution ahead of schedule. This robustness suggests that our solution could be effectively applied to different AI running at the edge.

Finally, our system's consistent performance across a wide range of conditions, its adaptability in algorithm selection, its robust and modifiable design, and its ability to meet diverse application requirements underscore its potential for broad application in the IoT field. The system's capacity to optimise operations based on each scenario's specific needs showcases its flexibility and potential for widespread adoption.

7.6 Conclusion

In the context of intelligent IoT applications, this chapter evaluates the implementation and performance of the system and scheduling approach discussed in *chapters 4, 5, and 6*. The primary objective is to examine the system's functionality across a variety of complex scenarios and evaluate its efficacy, adaptability, and viability with respect to the proposed scheduling approach.

In the smart city scenario, our system pairs seamlessly with an adaptive setup to enhance an expensive-to-run application. It successfully minimises the cost of running GPU-accommodating cloud instances, ensuring service continuity. GNH's efficacy is showcased in its rapid and reliable deployment of essential IoT applications, demonstrating resilience across diverse infrastructural circumstances.

In the context of agriculture, autonomous robots develop machine learning models collaboratively. Our system operates effectively under a decentralised control architecture, demonstrating its adaptability with individual controllers and shared fog nodes. EO-GNH outperforms GNH in federated learning deployment and decision-making, especially when more mappers are integrated. This scenario suggests the possibility of optimising resource use versus outcomes and offers opportunities for the improvement of decision-making strategies.

EO-GNH excels in deploying a temperature forecasting model within a smart factory environment, illustrating its proficiency in managing concurrent applications. Regardless of the scale, even up to 1000 locations, EO-GNH quickly identifies optimal

solutions. This capability is the result of a balanced approach to exploration and exploitation, a strategy that consistently yields high-quality results.

In conclusion, the chapter provides valuable insights into the performance of the system across a variety of applications. The findings highlight the adaptability of the system, the superiority of EO-GNH over GNH, and the potential for additional decision-making and scheduling strategy refinement. Future research could concentrate on optimising the EO-GNH oracle to better balance exploration and exploitation capabilities in accordance with the deployment requirements of IoT applications.

Chapter 8

Conclusions and Future Directions

8.1 Introduction

In this concluding chapter of the thesis, we provide a summary of our work and describe how we addressed complex challenges and research questions in IoT application management. We evaluate the contribution of our motivations, methodology, and key findings to the body of knowledge in this field. In particular, we have addressed IoT infrastructure and application decoupling, parallel programming, function placement, platform integration, and evaluation of placement quality.

In addition, we will investigate potential future research paths. As is typical in research, our investigation has uncovered new questions and methods for investigation. In order to continue advancing our understanding of IoT application management, we will identify areas for future research.

This final chapter not only provides a comprehensive summary of our work, but it also lays the groundwork for future research in this field.

8.2 Resolving Research Questions in IoT Application Placement

This section provides clear responses to the research questions presented in *Chapter 1*, encompassing key areas of the IoT system including application placement, infrastructure optimisation, platform integration, and scheduling strategies. The subsequent subsections look into each of these topics, providing detailed responses to the previously outlined research questions.

8.2.1 Modelling the Placement Problem

The first research question was centred on accurately modelling the placement problem in IoT applications, especially considering the complexities of a fog-cloud infrastructure. To address this, we employed integer linear programming (ILP), a method known for its ability to handle complex combinatorial optimisation problems, which we detailed in *Chapter 3*.

Our ILP model is not merely a mathematical abstraction of the placement problem. Instead, it provides a comprehensive perspective, encapsulating the infrastructure state, precise execution requirements, and deployment objectives. This holistic approach makes the model versatile and adaptable, suitable for various scenarios, and enables an improved scheduling and placement process.

Further, this model also incorporates a risk assessment formula and a cost reduction method to enhance the resilience of the management process. This is particularly important in dynamic IoT environments. A significant feature of our model is its allocation function, which favours early-stage service functions and strategically avoids high-risk locations while effectively utilising replicas to compensate for potential time losses in the event of failures.

Moreover, the model acknowledges that the costs of application deployment are directly related to the resources used during the application's execution. It incorporates these aspects to give a holistic understanding of the placement problem and the application of ILP techniques for resource optimisation.

By setting a solid foundation and providing a reliable roadmap for the optimisation algorithm, this model has proven its value by outperforming intelligent programmes with limited replicas in real-world testing and serving as a flexible framework for managing a wide array of IoT applications.

8.2.2 Defining a Platform Architecture

The second part of our research question was to define a platform architecture capable of effectively managing and operating IoT applications. As described in *Chapter 4*, we responded by developing an innovative platform based on a fog-cloud infrastructure. This platform includes an extensive toolkit for defining, monitoring, analysing, scheduling, and executing applications, making it a comprehensive solution for managing IoT applications.

A unique feature of this platform is its capacity to link platform tools with optimisation algorithms, enabling adaptive system management. This adaptability is particularly beneficial in the dynamic IoT environment, where changes in application requirements or infrastructure state can demand alterations to the placement strategy.

The architecture of the platform also supports customisation, allowing it to align with user-defined decision-making algorithms. It handles data transformations across various data models and transmission procedures and emphasises the importance of dynamic optimisation in platform operations.

Furthermore, it utilises forward dynamic programming techniques for managing the Service Function Chain (SFC) graph–an approach especially suited for real-time, event-based systems. The platform's architecture is thus designed to be robust, versatile,

and dynamic, meeting the immediate needs of managing IoT applications while also providing the flexibility to adapt and grow in response to future challenges.

For a comprehensive understanding of the decision-making process for SFC placement scheduling, we focused on creating scheduling algorithms for SFC placement in Chapters 5 and 6. The aim was to provide a detailed insight into the decision-making component of the platform.

8.2.3 Application Placement with Consideration for Multiple Objectives

This section of our thesis addresses the second research question, which involves the development of a strategy for IoT application placement that accounts for multiple objectives. To achieve this, we have introduced two novel approaches: the Greedy Nominator Heuristic (GNH) and the Enhanced Optimized-Greedy Nominator Heuristic (EO-GNH).

The GNH strategy, detailed in *Chapter 5*, is a greedy method that leverages the MapReduce paradigm to reduce end-to-end latency across a SFC. It accomplishes this by adhering to two strategies. First, GNH avoids deploying functions on unstable computing resources, i.e., those with a historically high failure rate. Second, it employs a replication strategy that takes into account the function's position in the SFC when deploying functions across multiple locations. Consequently, functions that appear earlier in the SFC get higher replication due to their direct impact on the execution of dependent downstream functions.

To enhance efficiency in multi-objective optimisation, GNH uses scalarisation, a process that consolidates several objective functions into one. Despite this advantage, GNH does not guarantee pareto front solutions – a drawback EO-GNH was designed to overcome.

EO-GNH, discussed in *Chapter 6*, is an optimisation algorithm that employs asynchronous MapReduce and parallel meta-heuristics to address GNH's limitations. It prioritises optimisation times, thereby enhancing the final outcome. The algorithm integrates the best-suited meta-heuristics for application scheduling and runs concurrently with the greedy approach, speeding up scheduling and improving placement quality.

In contrast to GNH, EO-GNH provides non-dominant solutions from the Pareto front, offering a more effective approach to multi-objective optimisation. This feature aids in identifying trade-offs among given solutions. Furthermore, unlike GNH which applies scalarisation in two phases, EO-GNH uses it once to extract one solution from the non-dominant ones. Combining this with meta-heuristics and a greedy approach leads to a more dynamic and responsive optimisation process.

While both GNH and EO-GNH contribute to efficient IoT application management, they have distinct capabilities and advantages due to their design differences. For instance, EO-GNH surpasses GNH in avoiding local optima and outperforms it in maintaining service availability on infrastructure with unreliable nodes. Specifically, EO-GNH achieves 100% service availability, compared to GNH's 95.5%. Thus, EO-GNH not only addresses GNH's limitations but also enhances the process of multi-objective optimisation, demonstrating the potential for further improvements in IoT application management.

8.2.4 Implementing Scheduling and Placement Strategy

The important objective of this research, which was addressed in Chapters 4, 5 and 6, was to devise and implement an effective scheduling and placement strategy that can optimise IoT applications running in fog-cloud environments. The strategy needed to be adaptive, leveraging dynamic programming and parallel computing for decision-making, while also factoring in the distinct characteristics of different applications and

the available infrastructure.

Chapter 4 concentrated on the decision-making aspect of scheduling, extending the decision-making component established earlier. The proposed strategy employed a greedy approach, utilising dynamic programming to traverse the SFC. This approach, built on the platform's dynamic nature, effectively examined the system's varying states and determined the optimal solution at each decision point.

Chapter 5 furthered this strategy by presenting the GNH, which utilises the parallel computing model MapReduce to accelerate the scheduling process. This enhanced method broke down the decision variable analysis into smaller, manageable tasks and executed them simultaneously, dramatically reducing the time required to determine an optimal placement plan.

Chapter 6 addressed the limitations of the MapReduce model and enhanced it using meta-heuristics. In parallel to this, a machine learning solution was developed that selects the most suitable meta-heuristic for an application based on its unique characteristics. This approach tackled two major issues simultaneously: accelerating the convergence to optimal solutions and choosing the most suitable meta-heuristic for a specific application.

Furthermore, an asynchronous version of MapReduce was introduced to handle the slowness of meta-heuristics. Each Mapper in this model continuously refines the pareto front, and a Reducer selects the best solution out of all pareto-front approximations provided by Mappers.

Chapter 7 demonstrated the versatility and efficacy of the proposed strategy by applying it to different IoT application domains, such as flood prediction in smart cities, temperature regulation in fish factories, and on-field weed identification in rural areas. The results confirmed that the proposed method can readily be combined with existing system parts to create a new, more flexible platform. The effectiveness of GNH in managing computationally intensive processes and working in harmony with other

systems was particularly notable. Implementing the proposed scheduling and placement strategy in real-world AI applications showcased its adaptability, robustness, and effectiveness in diverse scenarios.

The first application demonstrated how the system effectively adapted to a computationally intensive and expensive-to-run application. This showcased the strategy's ability to integrate with other adaptive systems to add new capabilities while managing costs efficiently. The reliable platform controlled the infrastructure under varying circumstances, while the GNH ensured rapid and dependable deployment of essential IoT applications.

The second scenario involved autonomous mobile robots working together to develop machine learning models. This application presented a decentralised control architecture, with each robot operating its own controllers and sharing fog nodes with other controllers. The strategy managed this decentralised structure efficiently, illustrating its adaptability to varying infrastructure layouts.

In the mobility simulation, the strategy's performance was evaluated using a federated learning deployment. The EO-GNH outperformed the GNH in all configurations. Adding more mappers improved decision-making and scheduling performance, although the improvements were minor. This experience suggested the possibility of a balance between the consumption of more resources for better outcomes and reducing the computational cost of decision-making. The EO-GNH's Oracle could be enhanced to determine the number of mappers, providing users with more control over resource allocation.

The successful deployment of the temperature forecasting application further underscored the effectiveness of EO-GNH's strategy. It outperformed GNH by delivering faster average completion times, even as the number of locations increased from 100 to 1000. This case brought to light the exploration and exploitation features of the mapper-level meta-heuristics and reducer-level greedy approach, respectively. It became evident that this configuration of exploration and exploitation is suitable for a

variety of decision-making scenarios

8.3 Solutions to Challenges

This section presents the solutions to the issues identified in *Chapter 1* across the various IoT system layers. We develop comprehensive solutions for significant problems in the application, infrastructure, platform, and scheduling layers.In the following subsections, the particulars of each solution that correspond to the previously outlined obstacles are discussed in depth.

8.3.1 Monitoring Infrastructure Changes and Utility Tools Integration

In managing the IoT infrastructure, we adopt an Object-Oriented Analysis and Design (OOAD) approach to effectively analyse and design complex systems. This approach is applied to monitor changes in infrastructure and integrate utility tools. It establishes new connections between various components and processes data for efficient decision-making, thereby improving the performance of the IoT infrastructure.

8.3.2 Tackling Unreliability and Failures

Our risk management approach addresses the challenges of unreliability and failures within IoT infrastructures. Monitoring task completion rates helps prevent function allocation to high-risk locations. Furthermore, a replication strategy provides redundancy for the application, effectively balancing the cost and performance of deploying replica functions, thereby mitigating the risk of failures.

8.3.3 Fostering Resource Awareness

We maintain consistent resource awareness across the system through effective monitoring, decision-making, and deployment components. Monitoring anticipates potential changes, decision-making analyses data and plans actions, while deployment shows real-time resource awareness during execution. This comprehensive resource awareness contributes to efficient management of dynamic IoT infrastructure.

8.3.4 Determining Adaptation Location

Our system excels at identifying the ideal location for the adaptation process, a critical factor determined by the system control type. This is achieved using the ILP model and OOAD principles. Real-world scenarios provide tangible examples of both centralised and decentralised control structures, illustrating the system's robustness and resilience in managing IoT infrastructure challenges.

8.3.5 Promoting Utility Tools Integration

The challenge of integrating utility tools is addressed using OOAD principles. This modular approach ensures seamless integration with various tools and future adaptability. The system's adaptability, facilitated by refined monitoring, decision-making, and deployment tools, enables integration with various software tools while effectively managing data transformations across different models and procedures.

8.3.6 Platform Integration and Self-Adaptive Features

We achieve integration with the IoT platform by combining Parsl with an optimisation tools, creating a responsive self-adaptive system to changing IoT conditions. The system components include deployments, monitoring, and decision-making, ensuring

the platform's adaptability to dynamic IoT environments. Our approach also facilitates the seamless integration of the decision-making component with other self-adaptive elements, resulting in a self-configurable, self-optimized, and self-healing system.

8.3.7 Evaluating Placement Quality Through Simulations

Simulations form a significant part of our research, providing a means to evaluate scheduling algorithms and understand complex IoT dynamics. Our custom simulation tools replicate real-world conditions, allowing for thorough testing of decision-making algorithms and balancing Quality of Service (QoS) metrics such as performance and cost.

8.3.8 Achieving Resource Accessibility

Resource accessibility is a critical requirement in complex computational tasks. We address this by employing Parsl [9], which manages resources effectively and rapidly executes decision-making and optimisation algorithms. Parsl also adjusts to various configurations, optimising resource use, and emphasising scalability. Thus, we provide a solution for effective resource control, utilisation, and scalability, even with resource expansion.

8.3.9 Service Function Chain Graph Design

The design of the SFC graph is crucial in IoT applications. We utilise Python's decorator pattern and functional programming within Parsl to manage this challenge effectively. An amalgamation of OOAD principles with functional programming creates distributed operators for task graph management, enhancing code readability and maintainability.

8.3.10 Decoupling Infrastructure and Application

Our work successfully decouples infrastructure and application by creating utility functions serving adaptive system components. The SFC graph is managed independently of infrastructure issues, simplifying debugging and ensuring smoother system operation. This decoupling enhances system flexibility and robustness.

8.3.11 Parallel Programming and Function Placement

We address the challenge of parallel programming and function placement

using the SFC graph encapsulated in Parsl apps and stored in the Python data model. The execute function of the deployment component with Parsl facilitates parallel task execution, improving resource utilisation, efficiency, and throughput. This effective use of advanced programming techniques results in adaptable, efficient, and resilient IoT systems.

8.4 Future Work

Future research and development opportunities in the field of IoT application deployment in fog-cloud environments are numerous. Several opportunities exist to improve the efficacy, adaptability, and strength of our strategy, building on the conclusions of this thesis. The sections that follow describe these potential research paths along with their respective objectives and anticipated outcomes, which aim to continue advancing this field toward greater efficiency and dependability.

8.4.1 Enhanced Financial Strategy for Application Management

In order to design a sustainable, cost-effective system for managing applications in edge-cloud infrastructures, our future direction involves developing a sophisticated financial strategy for IoT applications. This will require extending the MaxReplicas approach to consider the financial cost of running various types of applications, particularly those requiring high computational power, such as GPU-tagged tasks. By incorporating a more comprehensive economic strategy, we can achieve a better balance between cost and performance, which is crucial for the system's scalability and sustainability.

8.4.2 Integrating Machine Learning Pipelines in the EO-GNH Oracle

We realised that the current EO-GNH oracle would benefit from the integration of a machine learning pipeline (i.e., automated machine learning). This would involve the development of a mechanism for the oracle to construct and process new machine learning models from operational data. Future work would entail not only establishing the pipeline but also ensuring that the models can be updated based on new incoming data. This would lead to continuous learning and improvement for the oracle, enhancing the accuracy and efficiency of predictions over time.

8.4.3 Improving Mobility Simulation

Random walk approach has limitations, such as the reduced effectiveness of a random walk in certain spatial configurations and challenges in creating the optimal field coverage potentially requiring customised strategies for different scenarios [101]. Enhancing this model by integrating a mechanism for collecting and utilising information to inform the robot's movements could significantly improve efficiency. Our mobility simulation currently lacks the capability to accurately simulate vehicle-to-vehicle interactions. As future work, the simulation model should be improved to accommodate

different movement patterns and speeds of various types of vehicles. This enhancement will better mimic real-world conditions, resulting in more accurate and useful simulations for testing and development purposes. We aim to refine our models by incorporating realistic mobility patterns, focusing on geographical and behavioural factors. This enhancement will improve our simulations' relevance and predictive accuracy, ultimately benefiting mobile edge device operations.

8.4.4 Incorporation of Reinforcement Learning

With the rising trend of self-supervised and agent-based learning, we plan to explore the potential of reinforcement learning, temporal difference learning, and the Markov decision process in our future work. Implementing these methods as optimisation algorithms could significantly enhance the system's adaptability and performance. Furthermore, the application of reinforcement learning in risk management could allow the system to assess and respond to risks more effectively.

8.4.5 Managing Uncertainty

Real-world environments where IoT applications can be deployed are dynamic and unpredictable. Anomalies, outliers, or sudden operational changes can significantly affect system performance and reliability. Developing mechanisms to handle these uncertainties ensures the system remains resilient and maintains performance standards during the execution phase of tasks. Therefore, we also plan to focus on the development of mechanisms for managing uncertainty in our future work. This will involve the creation of algorithms or processes to detect anomalies, outliers, and sudden changes in the operational environment. Such capabilities will improve the system's resilience and adaptability in the face of real-world uncertainties.

8.4.6 Addressing Security in System Scalability

As the system scales to accommodate a growing number of edge devices, it is critical to address security concerns. Future work in this area will involve exploring safe and secure methods for sharing computational resources at the fog layers. This will require the development of a robust mechanism for establishing a trust chain in the infrastructure.

8.4.7 Exploring GPU-based Meta-heuristics

Using GPUs for running meta-heuristics is promising aspect because GPUs offer parallel processing capabilities, significantly accelerating computations. This is particularly beneficial for complex and time-intensive tasks in meta-heuristic algorithms, leading to faster solution finding and improved performance in handling large-scale and computationally demanding problems. The parallel processing capabilities of GPUs could be harnessed to reduce execution time and improve the quality of solutions in complex optimisation problems. Future efforts would also involve extending libraries to support GPU programming, enabling a wider range of meta-heuristics to be implemented on GPUs.

8.5 Concluding Remarks and Future Prospects

In this thesis, we have made significant advancements in the field of IoT application placement in fog-cloud environments, a challenging project requiring the application of complex algorithms and an in-depth understanding of system behaviour. Recognising the rapidly changing IoT landscape, we view our progress as a stepping stone on a much longer journey, not a final destination.

The future research directions stated previously highlight the vast number of upcoming opportunities. Integration of machine learning pipelines and reinforcement learning,

refinement of application financial management, and strengthening of the security aspects of scalable systems are just a few of the many areas requiring additional research.

We continually aim for innovation and examine existing models with an alternative viewpoint. We seek optimal placement solutions within the dynamic complexity of IoT applications. With the ultimate aim of creating more efficient, resilient, and adaptable systems, the current insights serve as a basis for further IoT research.

Appendix A

Background and Research Context

The system's main goal is to execute application functions on the infrastructure's process nodes. Under a dynamic infrastructure, an application's execution must meet QoS. The platform builds infrastructure-related knowledge to facilitate management decisions that satisfy QoS requirements. Figure A.1 gives an overview of the system components and illustrates how the system manages the applications operations in fog-cloud infrastructure.

An application is built from functions that have dependencies between them, and each function has its own requirements. Functions requirements can be software or hardware, such as packages or GPU. Alternately, it can be policy-based, such as level of privacy or execution deadline. QoS is defined as objectives that have quantifiable measures of performance. The source code of functions, or its executable program, has to be present when acting on input data. The programme is mostly embedded in the process node, and data is sent via network to be processed.

Infrastructure components are either process nodes or network components, and their characteristics affect the execution QoS. Infrastructure resources vary in network capability and node capacity and have different resource configurations. Node capacity is the available computing resources, such as memory or process type, whereas network capability is about transmission quality metrics. Node configuration is related to the software components that enable the application's execution. Communication configuration is about the arrangement that allows the communication to reach nodes.

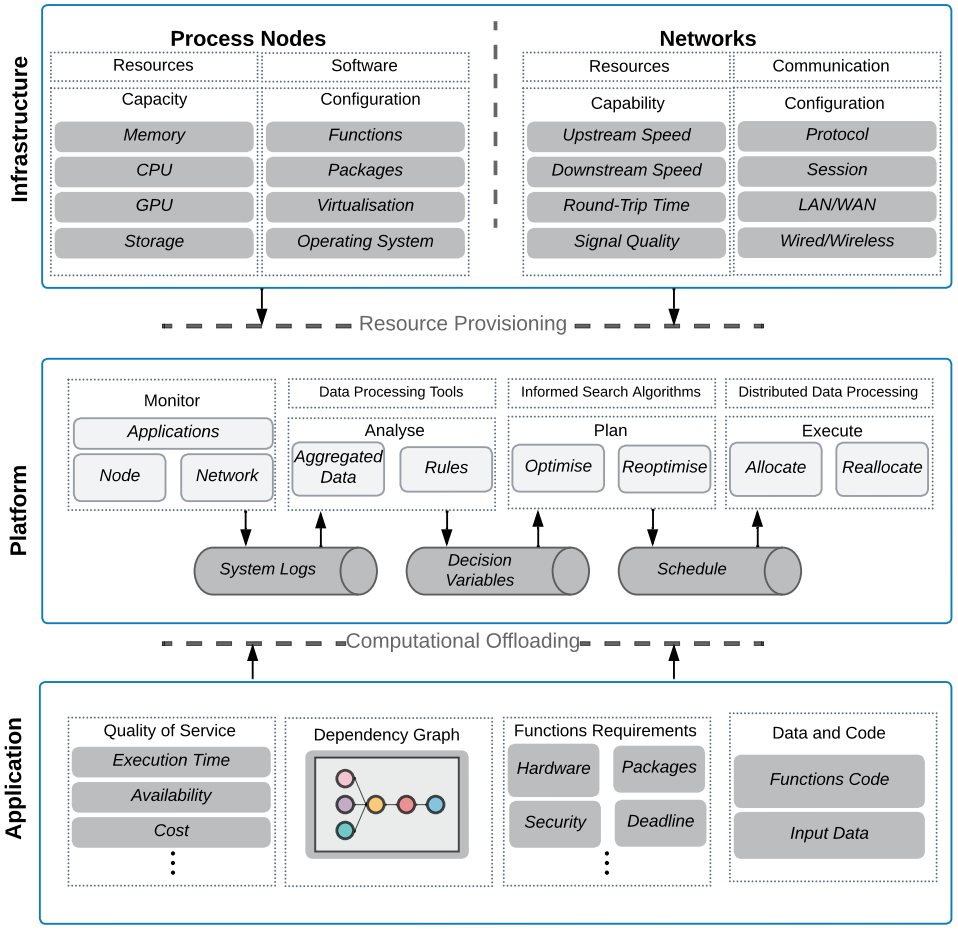


Figure A.1: Adaptive platform-based application and infrastructure management .

The platform manages the offloading process by building knowledge to execute applications. Then it uses analysis tools to create decision variables from the preliminary process system log. The decision variables are used by an algorithm to search for an optimal scheduling plan. Finally, the distributed data processing tools are utilised to allocate applications' functions in the process node for execution.

The following sections detail the characteristics of the IoT domains, adaptive platforms, and search algorithms.

A.1 Application Areas & Applications Attributes

A.1.1 IoT Domains and Applications

IoT solutions provide smart applications in many domains. Every IoT domain has a computing environment that shapes the application and infrastructure design and configuration [102]. IoT domains include smart homes, industrial IoT, smart healthcare, precision agriculture, etc. The role of IoT in various applications domains are shown in Table A.1.

The technology and the aim of an application govern its QoS, dependency design, function requirements, data type, and code base [103, 104, 105]. For instance, applications that collect and transmit data require a robust network and a location to store data, whereas machine learning applications require a GPU and a large amount of memory.

A.1.2 Application Layer Components

The components of the application layer allow the platform layer to orchestrate application executions. The code of the functions focuses on the transformations that apply to the input data in order to produce the desired output.

To execute a function in a process node, the node must have the minimum requirements to run the function. Searching for resources to execute functions depends on the characteristics of the infrastructure, which vary from one domain to another. Therefore, setting up QoS requirements as high-level goals that guide the orchestration of the applications. These variations and the associated Quality of Service (QoS) requirements for each domain will be explored in more detail in Section C. Table A.2 defines the objectives of applications.

A.1.3 Node Capacity and Configuration

In function execution, node capacity and configuration are crucial factors because they can affect the performance and dependability of the functions operating on the platform. Capacity refers to the available computing resources, such as memory and the CPU.

The configuration of process nodes refers to the setup required to execute a function. Function is the main component and programming language, and the software packages ensure services are running and producing the desired output. The type of virtualisation technology that runs each function controls the allocated node resources. Virtualisation components, such as containers, manage communication with the operating system kernel to request computing resources and to provide a consistent and dependable runtime environment.

A.1.4 Network Configuration and Capability

Network configuration and capability are essential for optimising the performance of the functions. The maximum data transfer rate between the edge and the process node is referred to as the upstream or downstream speed. Round-trip time is the time it takes for a request to be sent from an edge device to a processing node and for a response to be sent back to the edge device.

This can be affected by many factors, such as the strength of the wireless signal between the edge device and gateway, which indicates how stable the connection is. The configuration of the network affects the performance of the network. While wireless connections are more flexible and convenient, they are subject to interference and signal degradation. Whereas wired connections are faster and more stable, they are less convenient for mobile devices.

In a LAN, WiFi networks are the most common type of connection between devices

within range of an access point or gateway router. The communication protocol, such as HTTPS and SSH, has conventions that govern the way that data is exchanged between devices over a network. Protocols impact the performance and security of function executions.

The network session types, which are stateful and stateless, define the session's information maintained about the application's executions. In the stateful session, the platform remembers information about function executions and is usually addressed by the protocol.

Application	Description
Smart homes	Devices in a smart home are linked together, letting the user remotely control home features like home security, temperature, lighting, and a home theater. Smart home technology provides homeowners with convenience and cost savings [106].
Smart cities	These cities are equipped with devices such as surveillance cameras, automated transportation, and environmental monitoring. Smart cities are capable of resolving major issues that affect people, such as pollution, water flooding, and traffic monitoring [13].
Industrial IoT	Applying real-time analysis of data generated by industrial machinery. Smart machines communicate their findings with businesses to make better and faster decisions, such as in energy management and increased production and efficiency [14].
Smart healthcare	Uses wearables, smart rooms, and smart gadgets to enhance the convenience of patients and healthcare workers. Monitoring and analysing patients' information, including glucose level, heart rate, blood pressure, etc. Support hospital operations, such as scheduling nurses and facilities [15].
Precision agriculture	Increasing agricultural production while lowering operating costs. To optimise resource use, it uses sensors, and cameras that monitor and analyse crop and soil health, and environment. Also, Robots and drones cover areas perform farming task [98].
Smart vehicle	vehicles equipped with sensors, actuators, cameras, and GPS to log user activity. Monitoring and analysing support for transport automation aims to enhance driver experiences, vehicle maintenance, traffic management, and business assistance [16].

Table A.1: IoT applications domains

Objective	Description
Service availability	The percentage of time that a service function is available and accessible upon request. The system must be able to continue functioning in the face of disruptions such as network outages, hardware failures, and software bugs.
Completion time	Determine the optimal schedule and resource allocation needed to complete a project within a given timeframe. Involving allocating resources to execute the functions, with the aim of avoiding bottlenecks and completing execution as quickly as possible.
Deployment cost	The explicit cost is the money spent per usage of the service. Whereas implicit cost is the ongoing expense required to run the application. This includes the maintenance and wear on resources. Therefore, the system should avoid using capacity that greatly exceeds the required resources.
Energy consumption	Energy efficiency allocation can be optimised using efficient resources and optimised functions. Functions could have two versions of the program, one of which is optimised to reduce energy consumption.
Data security	The measures to protect data and ensure only authorised access to data. Applying data encryption, access controls, and incident response secures the application's execution.
Service scale	Automatically adjusts the resources required based on the volume of application requests. Invoking the process nodes or its virtual resources to handle the workload.
Application's throughput	The rate at which the platform processes application workloads in time units. This is affected by various measures, such as function performance, function concurrency, network latency, and data streaming processing tools.

Table A.2: QoS of IoT applications

A.1.5 Addressing IoT Requirements

Addressing the diverse requirements of IoT applications across various domains is critical for ensuring optimal performance and dependability of the system. Each domain's requirements encompass specific computing environments, robust network connections, adequate data storage solutions, appropriate node capacity and configuration, virtualisation technologies, and suitable communication protocols.

Furthermore, considering infrastructure configuration and employing suitable data streaming tools and frameworks can significantly impact the processing of dynamic data generated by IoT applications. The adoption of high-level programming abstractions and the implementation of adaptive approaches integrated with streaming data engines enable effective resource management and monitoring.

By addressing these requirements, IoT applications can achieve success in their deployment and operation across diverse application areas.

A.2 Tools Support Distributed Data Analysis

A.2.1 Overview of Streaming Data Engines Generations

The processing of dynamic data is an important and evolving field, and one of the key technologies that has emerged to handle this is streaming data engines. In a review conducted by de Assuncao et al. [8], multiple generations of data stream processing frameworks were examined and categorised into four distinct generations.

These generations are extensions to traditional database management systems, distributed execution, user-defined functions, and highly distributed edge-cloud computing. Each subsequent generation builds on the strengths of the previous one while addressing its limitations.

In Table A.3, examples of data streaming tools categorised by generations are presented.

Generation	Examples
Extensions to traditional DBMS	NiagaraCQ [107], Aurora [108], STREAM [109]
Distributed execution	Medusa [110], Borealis [111]
User-defined functions	Apache Storm [112], Twitter's Heron [113], Apache S4 [114], Apache Flink [115], Spark Streaming [116], Apache Oozie [117], Apache Azkaban [118], Apache Airflow [119]
Highly distributed service functions	Amazon Lambda, Google Cloud Functions, Azure Functions, Node-RED [120], OpenWhisk [121], OpenFaaS [122]

Table A.3: Generational categorisation of data streaming tools

A.2.2 Detailed Analysis of Streaming Data Engines Generations

The first generation of streaming data engines involves the extension of traditional database management systems. In this approach, data is stored on storage media and SQL-like languages are used to query unbounded data streams. Relational operations such as joins, aggregations, filtering, and analytics can be performed on table components.

The second generation involves distributed execution using parallel distributed computation, which allows for more efficient processing of large amounts of data. However, it has struggled with load balance and resource management.

The third generation of streaming data engines involves user-defined functions that operate on the dependency chain for tasks in graphs and starts user-defined functions to process incoming data streams. This approach allows for more flexibility in processing

data and performing complex operations. However, it lacks high-level programming abstractions and is not suitable for IoT applications because it depends on batch processing.

Finally, the fourth generation of streaming data engines is highly distributed edge-cloud computing. This approach is responsible for processing service function graphs (SFCs) which define the workflow of IoT applications and require low-latency processing. It introduces the need for high-level programming abstractions to make it easier for developers to create and deploy stream processing applications on highly distributed infrastructures.

A.2.3 Utilisation and Potential of Streaming Data Engines

The tools mentioned serve as a layer of software that facilitates the execution process and impacts the user's static configuration. However, despite their benefits, these tools and concepts currently lack decision-makers with a comprehensive understanding of infrastructural problems necessary for effective planning. Nonetheless, they enable monitoring and scheduling through the use of a baseline load balancer algorithm. As technology advances, these features can be used in an adaptive approach integrated with streaming data engines to effectively process dynamic data.

These tools facilitate our upcoming discussion on autonomic control and its phases. The planning phase, which is managed by the optimisation algorithm within the adaptive loop, enhances platform planning, while data streaming tools handle resource utilisation.

A.3 Autonomic Control: Phases and Processes

A.3.1 The Concept of Adaptive Systems in Autonomic Computing

In system control, a closed-loop process lies at the heart of adaptive systems [6]. This loop is known as the MAPE-K loop in the context of autonomic computing [12] and is also referred to as adaptation management [123]. It includes a series of procedures for implementing and accumulating observations, analysing them to plan modifications, and implementing adjustments. The autonomic control can be summarised into four phases: data collection, analysis, plan, and action [124].

Typically, the system manager is in charge of these phases. If we divide the system into two parts - the platform and the managed (infrastructure and applications) - the platform is then the component that adapts to changes in the managed and its surroundings [125]. Figure A.2 shows the adaptive process and how the system acquires knowledge about the managed components in order to operate the system optimally.

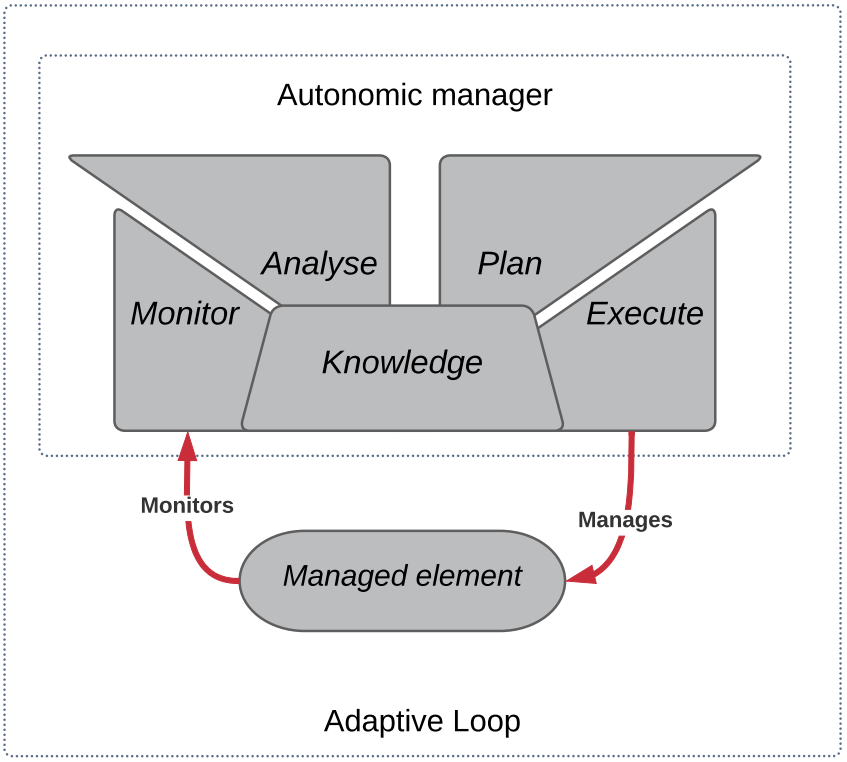


Figure A.2: Adaptive loop allows the system to be controlled under environmental changes .

A.3.2 Roles of the Platform Layer in Adaptive Processes

The platform layer in Figure A.1 serves as the managerial component that governs all adaptive processes in the system. The processes start with data collection, which involves monitoring the system components and tracking their state history to detect and respond to environmental changes through preparation or reaction.

Following data collection is the analysis phase, where the collected data is cleansed and encoded, and decision variables are determined. Subsequently, the decision-making process employs a strategy or algorithm to generate decisions based on the collected and analysed data.

During the decision-making and planning phase, the system determines whether to dy-

namically optimise or reoptimise in response to changes in the environment, or whether to adhere to the initial optimisation decision. Finally, during the acting phase, the system utilises data streaming tools to allocate or reallocate functions, thereby deploying the applications.

A.4 Search Algorithms for scheduling

A.4.1 The Importance of Efficient Scheduling and Informed Search

Efficient scheduling is essential for various applications and systems, involving organising tasks and allocating resources to ensure optimal utilisation and timely completion. It involves searching for scheduling solutions out of a set of possible solutions known as the search space [17].

The search space is the set of all possible solutions to a problem, and optimisation algorithms aim to identify the best solution within this space. The search space can be vast and complex, posing challenges to the optimisation process. Informed search algorithms play a pivotal role in enhancing the scheduling of tasks, leading to better performance.

Informed search explores the search space using heuristic functions and objective functions to gauge and evaluate their quality based on specific criteria. A heuristic function is typically employed to drive the search process toward the global optimum, but it will almost always result in a good approximation [18]. This improves efficiency in finding the best or near-best solution.

A.4.2 Structural Components of Optimisation Algorithms

The structure of an algorithm is defined by several key components that contribute to its effectiveness and efficiency. Figure A.3 summarises the structure of optimisation

algorithms.

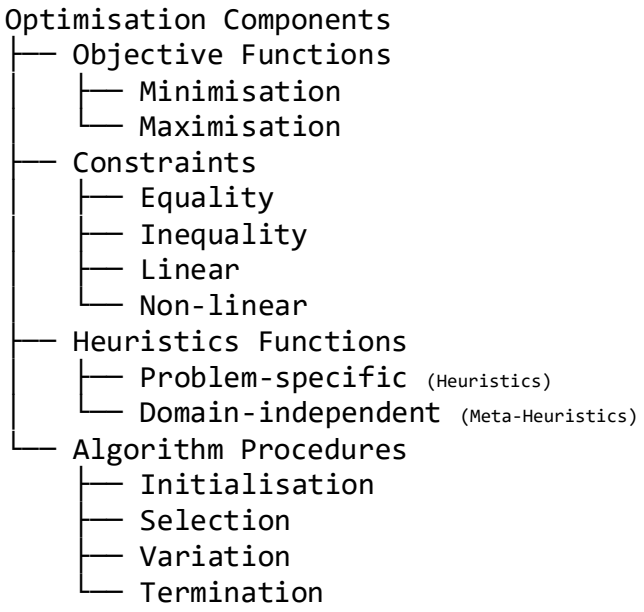


Figure A.3: Algorithm components

Objective functions and constraints define the problem and the algorithm that optimises the solution. The objective functions are the central aspect, representing the function that the algorithm aims to optimise by either minimising or maximising its value. Constraints are also essential, imposing conditions or restrictions that candidate solutions must satisfy to be considered feasible.

Algorithm procedures are essential steps in optimisation algorithms that guide the search process towards optimal or near-optimal solutions. They involve selection and variation mechanisms. Selection prioritises promising solutions based on their quality, while variation generates new candidate solutions by modifying selected previous solutions [19]. The combination of these procedures ensures a balance between maintaining promising solutions and exploring the search space to avoid local optima and converge to global optima (balance between exploring and exploiting). Heuristics play

a crucial role in this structure, as they optimise solutions for complex problems by utilising domain knowledge and insights to create rules or guidelines that guide the search towards promising areas in the search space. Integrating heuristics into the design of an algorithm enhances the optimisation process's effectiveness and efficiency.

A.4.3 Properties of Multi-objectives Optimisation Algorithms

The properties of multi-objectives optimisation algorithms significantly impact the quality of solutions, making it essential to consider them when designing optimisation methods. Figure A.4 summarises the properties.

These properties include preference information, which involves the various types of preference data provided by a user or decision-maker to the optimisation's goals, such as a priori, progressive, or a posteriori preference information, or even no articulated preference information [20].

Solution evaluation encompasses methods used to assess and compare the quality of solutions within the search space, employing techniques like scalarisation and Pareto optimality [21]. In multi-objective optimisation, scalarisation aggregates multiple objective functions into a single one, while Pareto methods prioritise non-dominated solutions across all criteria among other explored alternatives. A set of non-dominant solutions offers a variety of potential alternatives, each with its own unique trade-offs; none of them is better than the other in all objectives [21].

Local and global optima are two important concepts in the field of optimisation.Local optima refer to the optimal solutions within a limited area or region of the search space, while global optima refer to the optimal solutions throughout the entire search space.Noting that a local optimum is not necessarily a global optimum is essential, as other regions of the search space may contain better solutions [17].

Exploration and exploitation are fundamental to the optimisation process.Exploration is the process of discovering new regions of the search space, which may provide

global optimums by revealing diverse solutions. Exploitation, on the other hand, entails augmenting the currently best-known solutions and refining them to discover local optima within a particular region of the search space. It is essential to find a balance between these two strategies; excessive exploitation could cause the algorithm to become trapped in suboptimal solutions, while excessive exploration could prevent the algorithm from adequately refining promising solutions. Therefore, a successful optimisation procedure must navigate carefully between exploration and exploitation [17].

The execution approach pertains to how the optimisation algorithm is carried out, incorporating parallelism and serial execution, where parallel algorithms can accelerate the search process while serial algorithms execute tasks sequentially [22].

Solutions quantity outlines methods used to navigate the search space, categorising optimisation algorithms based on the number of concurrent solutions, including single solution-based algorithms (trajectory methods) and population-based algorithms that operate on a set of solutions simultaneously [19].

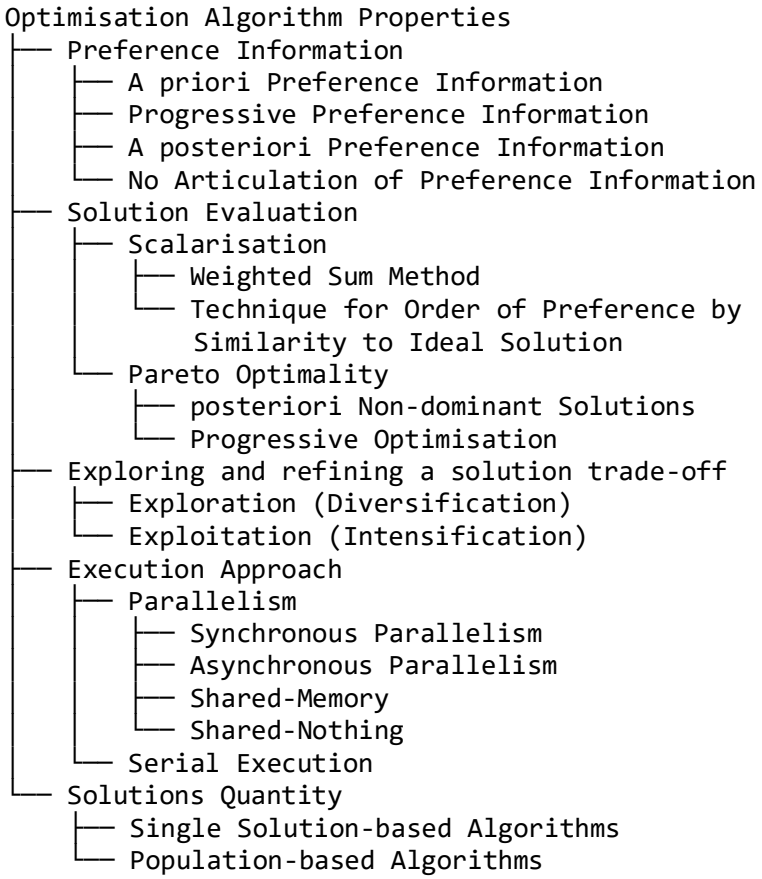


Figure A.4: Algorithm properties

Appendix B

Systematic Review Process

B.1 The Choice and Role of Search Engines in the Review

The systematic review is based on a semi-automatic technique where key extraction and topic modelling tools were utilised along with several search engines. We employ a web of science search engine during the investigation.

We pick this search engine since it permits search over numerous scientific bodies such as IEEE, AMC and Springer. Also, the web of science allows readers to readily access a citation tree, where the citation link between works may be accessed with simplicity.

B.2 Parameters for Publication Selection

Explore publications published between January 2016 and March 2022. Keywords include synonyms for optimising the placement of IoT application activities in edge, fog and cloud contexts utilising meta-heuristics and heuristics-based techniques. IoT, Fog, Edge, Cloud, offloading, placement, deployments, meta-heuristics, heuristics, optimisations, search, smart city, smart factory, etc. are examples of keywords.

This survey did not include articles that were unrelated to work scheduling or employed a purely machine learning methodology (papers that use a hybrid approach

were included).

B.3 The Role of Natural Language Processing Tools

We implement a search engine based on text indexing, keyword extraction, and subject classification using a variety of NLP tools. Natural Language Toolkit (NLTK) [126] builds an index-based search engine with classification, tokenisation, stemming, and labelling capabilities. Keyword extraction is handled by two unsupervised methods, YAKE [127] and RaKUn [128], with YAKE being a corpus-independent tool and RaKUn utilising graph-based language representations to perform rank-based extraction quickly. In addition, NLTK's stemming indexing capability manages repetitive words, thereby improving search precision. These methods are not employed individually. Instead, they collectively contribute to the final assessment, supplying valuable insights even when the authors of the paper have not expressly stated certain information.

B.4 The Step-by-Step Process of the Survey

The process of conducting a systematic review involves multiple phases. Initially, search engines are used to locate research papers based on particular keywords. These articles' titles and abstracts are then stored in a structured database for simple access and organisation.

In the second phase, abstracts of these publications are filtered using Natural Language Processing (NLP) tools. This stage functions to eliminate irrelevant articles while retaining potential candidate papers for further review. The selected documents are then converted from PDF to text using Apache Tika. This preparation phase is essential because it prepares the textual data for use with NLP tools.

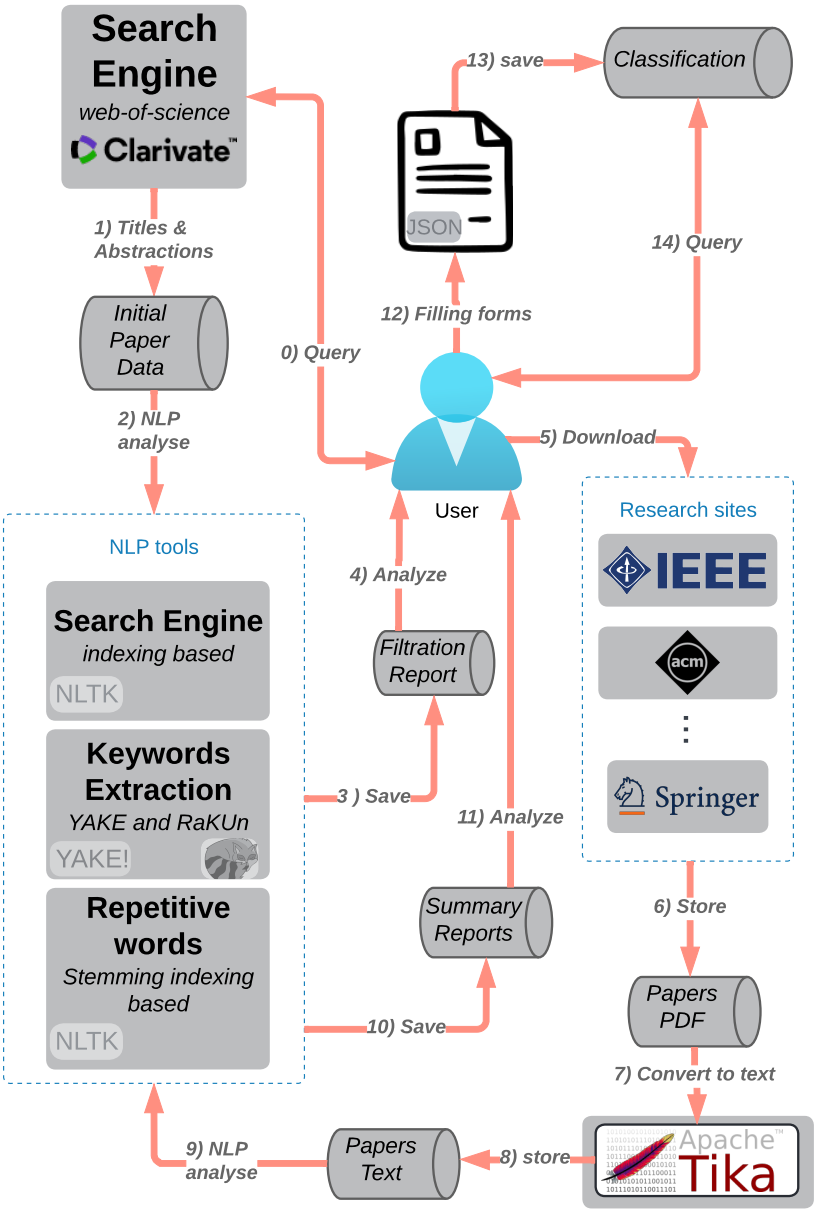


Figure B.1: Systematic review steps and processes

The processed text is then analysed with NLP tools, which, along with a comprehensive reading of the papers, helps in their categorisation. The content of each document is systematically classified and summarised using a form template. The procedure for conducting a systematic review is illustrated in Figure B.1.

Appendix C

Survey Results

C.1 Detailed Overview of Optimisation Algorithms

Table C.1 (GA Algorithm References): Presents applications of the Genetic Algorithm (GA) in different fields such as Vehicle, Infrastructure, Industry, City, etc., listing the referenced papers for each application type.

Application Type	**Referenced Papers**
Vehicle	[29], [34]
Infrastructure	[29], [129], [130], [33], [131], [132], [133], [56], [134], [54], [47], [135]
Industry	[130], [136], [38], [54], [137], [47], [135], [25]
City	[42], [28], [48], [55]
Home	[138]
Healthcare	[139]
DNN	[140]
Robot	[136], [137]
Unspecified Domain	[36], [141], [37], [56], [142], [143], [134], [144], [44], [145], [43], [146], [62]
Image processing	[58]
Signal processing	[58]
Data analysis	[58]
Agriculture	[147]
Information retrieval	[43]

Table C.1: GA algorithm references

Table C.2 (PSO and ACO Algorithm References): Details the applications of the Particle Swarm Optimisation (PSO) and Ant Colony Optimisation (ACO) algorithms across different sectors, again referencing relevant papers.

Algorithm	Application Type	Referenced Papers
PSO	Infrastructure	[131], [56], [46]
	Industry	[38], [148], [39], [149], [150]
	City	[151], [46]
	Home	[138]
	DNN	[140]
	Robot	[150]
	Unspecified Domain	[56], [152], [153]
	Image Processing	[46]
ACO	Vehicle	[154]
	Industry	[35], [149]
	City	[151]
	Home	[138]
	Healthcare	[139]
	DNN	[35]
	Unspecified Domain	[45], [155]

Table C.2: PSO and ACO Algorithm References

Table C.3 (Greedy and RB Algorithm References): Shows the use of Greedy algorithms and Rule-Based (RB) algorithms in various contexts like Infrastructure, Industry, City, and IoT, with citations for each application.

Algorithm	Application Type	Referenced Papers
Greedy	Infrastructure	[31], [49], [40], [51], [53], [54]
	Industry	[54]
	City	[50], [53], [55]
	General IoT	[40], [52]
RB	Vehicle	[156], [157]
	Infrastructure	[32], [158], [159], [41], [160], [53]
	Industry	[41]
	City	[156], [160], [161], [53]
	Healthcare	[162], [163], [164]
	DNN	[165], [166]
	General IoT	[167], [168], [169], [52]

Table C.3: Greedy and RB Algorithm References

Table C.4 (Summary of Algorithms and their Applications): This table summarises different algorithms like Neural Networks (NN), Constraint Satisfaction Problem (CSP), Simulated Annealing (SA), etc., and their applications in fields such as City, Home, Vehicle, etc., providing references for each algorithm-application pair.

C.2 Comprehensive Overview: Table

Table C.5 presents a comprehensive overview of various properties related to the optimisation algorithms, covering process (parallel or serial), solution quantity (single or population-based), preference (posteriori, progressive, or priori), search (trade-off exploration and exploitation), evaluation (non-Pareto or Pareto), and adaptiveness (static or dynamically reoptimising scheduling plan).

Application	**Algorithm**	**Referenced Papers**
City	NN	Memari et al. [57]
	CSP	Kamal et al. [156], Naas et al. [161]
	Relaxation	Bolettieri et al. [170]
	GP	Naas et al. [161]
Home	NN	Memari et al. [57]
Vehicle	CSP	Kamal et al. [156]
DNN	CSP	Hadidi et al. [166]
General IoT	SA	Najafizadeh et al. [171]
	AHP	Morkevicius et al. [152]
	MPA	Abdel-Basset et al. [172]
	WOA	Paul Martin et al. [173]
	MA	Sami and Mourad [62]
	GSA	Karamoozian et al. [26]
	SE	Tsai [61]
Healthcare	Gradient	Zhao [174]
	PeSOA	Benamer et al. [164]
	FF	Lin et al. [175]
Infrastructure	SE	Tsai [61]
	KH	Yang et al. [27]
ML	Gradient	Zhao [174]

Table C.4: Summary of Algorithms and their Applications

Table C.5: Survey of Optimisation for scheduling

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Ouedraogo et al. [29]	Serial	Population	GA, DP	Priori, Posteriori	Balanced	Pareto	Static
Huang et al. [30]	Serial	Single	DP	Priori	Exploitation	Non-Pareto	Static
Qu et al. [31]	Parallel	Single	Greedy	Priori	Exploitation	Non-Pareto	Dynamic
Mouradian et al. [49]	Serial	Single	Greedy, Tabu	Priori	Balanced	Non-Pareto	Dynamic
Gamal et al. [129]	Serial	Population	GA	Priori	Balanced	Pareto	Static
Yang et al. [32]	Parallel	Single	RB	Priori	Exploration	Non-Pareto	Dynamic
Tout et al. [130]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Memari et al. [57]	Serial	Single	Tabu, NN	Priori	Balanced	Non-Pareto	Static
Khaleel and Zhu [158]	Serial	Single	RB	Priori	Balanced	Non-Pareto	Static
de Freitas Bezerra et al. [33]	Serial	Population	GA	Posteriori	Balanced	Pareto	Static
Zhang et al. [159]	Serial	Single	RB	Priori	Exploitation	Non-Pareto	Dynamic
Kamal et al. [156]	Parallel	Single	RB, CSP	Priori	Balanced	Non-Pareto	Static
Yadav et al. [131]	Serial	Population	GA, PSO	Priori	Balanced	Non-Pareto	Static
Mishra et al. [148]	Serial	Population	PSO	Priori	Balanced	Non-Pareto	Static
Ghosh and Simmhan [132]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Dynamic
Gu et al. [162]	Parallel	Single	RB,	Priori	Exploitation	Non-Pareto	Dynamic

Table C.5 – continued from previous page

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Chen et al. [140]	Serial	Population	GA, PSO	Priori	Balanced	Non-Pareto	Static
Mohamadi et al. [34]	Serial	Population	GA	Progressive	Balanced	Pareto	Dynamic
Huang et al. [35]	Parallel	Population	ACO	Posteriori	Balanced	Pareto	Static
Xie et al. [136]	Serial	Population	GA	Posteriori	Balanced	Pareto	Static
Taghizadeh et al. [36]	Parallel	Single	GA	Posteriori	Balanced	Pareto	Static
Peng et al. [141]	Parallel	Population	GA	Posteriori	Balanced	Pareto	Static
Najafizadeh et al. [171]	Serial	Population	SA,	Posteriori	Balanced	Pareto	Static
Morkevicius et al. [152]	Serial	Population	PSO, AHP	Posteriori	Balanced	Pareto	Static
Maia et al. [37]	Parallel	Population	GA	Posteriori	Balanced	Pareto	Static
Hao et al. [133]	Serial	Single	GA	Posteriori	Balanced	Pareto	Static
Abdel-Basset et al. [172]	Serial	Population	MPA	Priori	Balanced	Non-Pareto	Static
Shahryari et al. [56]	Parallel	Population	GA, PSO	Priori	Balanced	Non-Pareto	Static
Bolettieri et al. [170]	Serial	Single	Relaxation	Priori	Exploitation	Non-Pareto	Static
Aburukba et al. [142]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Dynamic
Wu et al. [58]	Serial	Population	GA, FL	Posteriori	Balanced	Pareto	Static
Natesha and Guddeti [143]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Ayoubi et al. [134]	Parallel	Population	GA	Posteriori	Balanced	Pareto	Dynamic

Table C.5 – continued from previous page

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Eyckerman et al. [154]	Serial	Population	ACO	Priori	Balanced	Non-Pareto	Dynamic
Zhao [174]	Parallel	Single	Gradient	Priori	Balanced	Non-Pareto	Dynamic
Abdelmoneem et al. [163]	Serial	Single	RB	Priori	Balanced	Non-Pareto	Dynamic
Aburukba et al. [144]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Dynamic
Fang et al. [147]	Serial	Population	GA	Posteriori	Balanced	Pareto	Dynamic
Paul Martin et al. [173]	Serial	Population	WOA	Posteriori	Balanced	Pareto	Static
Javanmardi et al. [165]	Serial	Single	RB	Priori	Balanced	Non-Pareto	Static
Hadidi et al. [166]	Serial	Single	RB, CSP	Priori	Exploitation	Non-Pareto	Static
Sami and Mourad [62]	Serial	Population	MA, GA, LS	Posteriori	Exploitation	Non-Pareto	Static
Cui et al. [41]	Serial	Single	RB	Priori	Exploitation	Non-Pareto	Static
Xu et al. [42]	Serial	Population	GA	Posteriori	Balanced	Pareto	Static
Benamer et al. [164]	Serial	Single	RB, Pesoa	Priori	Balanced	Non-Pareto	Dynamic
Gao et al. [59]	Serial	Single	LO	Priori	Balanced	Non-Pareto	Dynamic
Huang et al. [28]	Parallel	Population	GA	Posteriori	Balanced	Pareto	Static
Hussein and Mousa [151]	Serial	Population	PSO, ACO	Priori	Balanced	Non-Pareto	Static
Vijouyeh et al. [160]	Parallel	Single	RB	Priori	Exploitation	Non-Pareto	Dynamic
Wang et al. [46]	Serial	Population	PSO	Priori	Balanced	Non-Pareto	Static

Table C.5 – continued from previous page

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Huang et al. [40]	Serial	Single	Greedy	Priori	Exploitation	Non-Pareto	Static
Fan et al. [45]	Serial	Population	ACO	Priori	Balanced	Non-Pareto	Static
Cui et al. [44]	Parallel	Single	GA	Posteriori	Balanced	Pareto	Static
Yousefpour et al. [50]	Serial	Single	Greedy	Priori	Exploitation	Non-Pareto	Dynamic
Nguyen et al. [145]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Misra and Saha [51]	Serial	Single	Greedy	Priori	Exploitation	Non-Pareto	Dynamic
Hajeer and Dasgupta [43]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Shao et al. [38]	Serial	Population	GA, PSO	Priori	Balanced	Non-Pareto	Static
Shao et al. [39]	Parallel	Population	PSO	Priori	Balanced	Non-Pareto	Dynamic
Djemai et al. [153]	Serial	Population	PSO	Priori	Balanced	Non-Pareto	Static
Karamoozian et al. [26]	Serial	Population	GSA	Priori	Balanced	Non-Pareto	Static
Moallemi et al. [48]	Parallel	Population	GA	Posteriori	Balanced	Pareto	Static
Wu and Wang [52]	Serial	Population	Greedy, RB, LS	Priori	Balanced	Non-Pareto	Static
Gravalos et al. [53]	Parallel	Single	Greedy, RB	Priori	Exploitation	Non-Pareto	Static
Ashok et al. [157]	Parallel	Single	RB	Priori	Balanced	Non-Pareto	Dynamic
Tsai [61]	Parallel	Population	LS, SE, K-means	Priori	Balanced	Non-Pareto	Dynamic
Naas et al. [161]	Serial	Population	RB, CSP, GP	Priori	Balanced	Non-Pareto	Static

Table C.5 – continued from previous page

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Yang et al. [27]	Parallel	Population	KH	Priori	Balanced	Non-Pareto	Dynamic
Skarlat et al. [54]	Serial	Population	GA, Greedy	Priori	Balanced	Non-Pareto	Static
Lyu et al. [60]	Serial	Single	LO	Priori	Balanced	Non-Pareto	Dynamic
Fan et al. [155]	Parallel	Population	ACO	Priori	Balanced	Non-Pareto	Dynamic
Naas et al. [167]	Serial	Single	RB	Priori	Balanced	Non-Pareto	Static
Rahbari et al. [138]	Serial	Population	GA, PSO, ACO	Priori	Balanced	Non-Pareto	Static
Rullo et al. [168]	Serial	Single	RB	Priori	Balanced	Non-Pareto	Static
Pham and Huh [169]	Serial	Single	RB	Priori	Exploitation	Non-Pareto	Static
Kuang et al. [55]	Serial	Population	GA, Greedy	Priori	Balanced	Non-Pareto	Static
Afrin et al. [137]	Serial	Population	GA	Posteriori	Balanced	Pareto	Static
Wang and Li [149]	Serial	Population	PSO, ACO	Priori	Balanced	Non-Pareto	Static
Wan et al. [150]	Serial	Population	PSO	Priori	Balanced	Non-Pareto	Dynamic
Gong et al. [47]	Parallel	Population	GA	Priori	Balanced	Non-Pareto	Dynamic
Kaur et al. [135]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Tang et al. [25]	Serial	Population	GA	Priori	Balanced	Non-Pareto	Static
Sun et al. [146]	Serial	Population	GA	Posteriori	Balanced	Pareto	Static
Lin et al. [175]	Serial	Population	FF	Priori	Balanced	Non-Pareto	Static

Table C.5 – continued from previous page

Paper	Process	Solution	Algorithm	Preference	Search	Evaluate	Adapte
Yu et al. [139]	Serial	Population	GA, ACO	Priori	Balanced	Non-Pareto	Static

Bibliography

[1] Fog Computing. the internet of things: Extend the cloud to where the things are. *Cisco White Paper*, 2015.

[2] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, 3:134–155, 2018.

[3] Paul Quinn and A Beliveau. Service function chaining (sfc) architecture. *draft-quinn-sfc-arch-04 (work in progress)*, 2014.

[4] Juliver Gil Herrera and Juan Felipe Botero. Resource allocation in nfv: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13 (3):518–532, 2016.

[5] Paul Quinn and Tom Nadeau. Problem statement for service function chaining. Technical report, 2015.

[6] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):1–42, 2009.

[7] Mohamed Boucadair et al. Service function chaining (sfc) control plane components & requirements. *Work in Progress, draft-ietf-sfc-control-plane-08*, 2016.

[8] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[9] Yadu N Babuji, Kyle Chard, Ian T Foster, Daniel S Katz, Mike Wilde, Anna Woodard, and Justin M Wozniak. Parsl: Scalable parallel scripting in python. In *IWSG*, 2018.

[10] Christian Berger, Philipp Eichhammer, Hans P Reiser, Jörg Domaschka, Franz J Hauck, and Gerhard Habiger. A survey on resilience in the iot: Taxonomy, classification, and discussion of resilience mechanisms. *ACM Computing Surveys (CSUR)*, 54(7):1–39, 2021.

[11] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future generation computer systems*, 56:684–700, 2016.

[12] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[13] Saber Talari, Miadreza Shafie-Khah, Pierluigi Siano, Vincenzo Loia, Aurelio Tommasetti, and João PS Catalão. A review of smart cities based on the internet of things concept. *Energies*, 10(4):421, 2017.

[14] Muhammad Bilal, Lukumon O Oyedele, Junaid Qadir, Kamran Munir, Saheed O Ajayi, Olugbenga O Akinade, Hakeem A Owolabi, Hafiz A Alaka, and Maruf Pasha. Big data in the construction industry: A review of present status, opportunities, and future trends. *Advanced engineering informatics*, 30 (3):500–521, 2016.

[15] Farzad Samie, Lars Bauer, and Jörg Henkel. Iot technologies for embedded computing: A survey. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2016.

[16] Fangchun Yang, Jinglin Li, Tao Lei, and Shangguang Wang. Architecture and key technologies for internet of vehicles: a survey. *Journal of Communications and Information Networks*, 2(2):1–17, 2017.

[17] David Meignan, Sigrid Knust, Jean-Marc Frayret, Gilles Pesant, and Nicolas Gaud. A review and taxonomy of interactive optimization methods in operations research. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(3):1–43, 2015.

[18] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984.

[19] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3): 268–308, 2003.

[20] Ching-Lai Hwang, Sudhakar R Paidy, Kwangsun Yoon, and Abu Syed Md Masud. Mathematical programming with multiple objectives: A tutorial. *Computers & Operations Research*, 7(1-2):5–31, 1980.

[21] Nyoman Gunantara. A review of multi-objective optimization: Methods and its applications. *Cogent Engineering*, 5(1):1502242, 2018.

[22] Enrique Alba. *Parallel metaheuristics: a new class of algorithms*. John Wiley & Sons, 2005.

[23] Claudia Canali and Riccardo Lancellotti. Gasp: genetic algorithms for service placement in fog computing systems. *Algorithms*, 12(10):201, 2019.

[24] Claudia Canali and Riccardo Lancellotti. A fog computing service placement for smart cities based on genetic algorithms. In *CLOSER*, pages 81–89, 2019.

[25] Chaogang Tang, Xianglin Wei, Shuo Xiao, Wei Chen, Weidong Fang, Wuxiong Zhang, and Mingyang Hao. A mobile cloud based scheduling strategy for industrial internet of things. *IEEE Access*, 6:7262–7275, 2018.

[26] Amir Karamoozian, Abdelhakim Hafid, and El Mostapha Aboulhamid. On the fog-cloud cooperation: How fog computing can address latency concerns of iot applications. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 166–172. IEEE, 2019.

[27] Yibo Yang, Yongkui Ma, Wei Xiang, Xuemai Gu, and Honglin Zhao. Joint optimization of energy consumption and packet scheduling for mobile edge computing in cyber-physical networks. *IEEE Access*, 6:15576–15586, 2018.

[28] Hualong Huang, Kai Peng, and Xiaolong Xu. Collaborative computation offloading for smart cities in mobile edge computing. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 176–183. IEEE, 2020.

[29] Clovis Anicet Ouedraogo, Samir Medjiah, Christophe Chassot, Khalil Drira, and Jose Aguilar. A cost-effective approach for end-to-end qos management in nfv-enabled iot platforms. *IEEE internet of things journal*, 8(5):3885–3903, 2020.

[30] Meitian Huang, Weifa Liang, Xiaojun Shen, Yu Ma, and Haibin Kan. Reliability-aware virtualized network function services provisioning in mobile edge computing. *IEEE Transactions on Mobile Computing*, 19(11):2699–2713, 2019.

[31] Long Qu, Chadi Assi, Maurice J Khabbaz, and Yinghua Ye. Reliability-aware service function chaining with function decomposition and multipath routing. *IEEE Transactions on Network and Service Management*, 17(2):835–848, 2019.

[32] Binxu Yang, Zichuan Xu, Wei Koong Chai, Weifa Liang, Daphné Tuncer, Alex Galis, and George Pavlou. Algorithms for fault-tolerant placement of stateful virtualized network functions. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2018.

[33] Diego de Freitas Bezerra, Guto Leoni Santos, Glauco Gonçalves, André Moreira, Leylane Graziele Ferreira da Silva, Élisson da Silva Rocha, Maria Valéria Marquezini, Judith Kelner, Djamel Sadok, Amardeep Mehta, et al. Optimizing nfv placement for distributing micro-data centers in cellular networks. *The Journal of Supercomputing*, pages 1–25, 2021.

[34] Houssem Eddine Mohamadi, Nadjia Kara, and Mohand Lagha. Efficient algorithms for decision making and coverage deployment of connected multi-low-altitude platforms. *Expert Systems with Applications*, 184:115529, 2021.

[35] Tiansheng Huang, Weiwei Lin, Chennian Xiong, Rui Pan, and Jingxuan Huang. An ant colony optimization-based multiobjective service replicas placement strategy for fog computing. *IEEE Transactions on Cybernetics*, 2020.

[36] Jaber Taghizadeh, Mostafa Ghobaei-Arani, and Ali Shahidinejad. A metaheuristic-based data replica placement approach for data-intensive iot applications in the fog computing environment. *Software: Practice and Experience*, 2021.

[37] Adyson M Maia, Yacine Ghamri-Doudane, Dario Vieira, and Miguel Franklin de Castro. An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in edge computing. *Computer Networks*, 194:108146, 2021.

[38] Yanling Shao, Chunlin Li, Zhao Fu, Leyue Jia, and Youlong Luo. Cost-effective replication management and scheduling in edge computing. *Journal of Network and Computer Applications*, 129:46–61, 2019.

[39] Yanling Shao, Chunlin Li, and Hengliang Tang. A data replica placement strategy for iot workflows in collaborative edge and cloud environments. *Computer Networks*, 148:46–59, 2019.

[40] Tiansheng Huang, Weiwei Lin, Yin Li, LiGang He, and ShaoLiang Peng. A latency-aware multiple data replicas placement strategy for fog computing. *Journal of Signal Processing Systems*, 91(10):1191–1204, 2019.

[41] Laizhong Cui, Shu Yang, Ziteng Chen, Yi Pan, Zhong Ming, and Mingwei Xu. A decentralized and trusted edge computing platform for internet of things. *IEEE Internet of Things Journal*, 7(5):3910–3922, 2019.

[42] Xiaolong Xu, Xihua Liu, Zhanyang Xu, Fei Dai, Xuyun Zhang, and Lianyong Qi. Trust-oriented iot service placement for smart cities in edge computing. *IEEE Internet of Things Journal*, 7(5):4084–4091, 2019.

[43] Mustafa Hajeer and Dipankar Dasgupta. Handling big data using a data-aware hdfs and evolutionary clustering technique. *IEEE Transactions on Big Data*, 5 (2):134–147, 2017.

[44] Laizhong Cui, Chong Xu, Shu Yang, Joshua Zhexue Huang, Jianqiang Li, Xizhao Wang, Zhong Ming, and Nan Lu. Joint optimization of energy consumption and latency in mobile edge computing for internet of things. *IEEE Internet of Things Journal*, 6(3):4791–4803, 2018.

[45] Jianhua Fan, Xianglin Wei, Tongxiang Wang, Tian Lan, and Suresh Subramaniam. Churn-resilient task scheduling in a tiered iot infrastructure. *China Communications*, 16(8):162–175, 2019.

[46] Xiaohui Wang, Hao Zhang, and Haoran Gu. Solving optimal camera placement problems in iot using lh-rpso. *IEEE Access*, 8:40881–40891, 2019.

[47] Xu Gong, David Plets, Emmeric Tanghe, Toon De Pessemier, Luc Martens, and Wout Joseph. An efficient genetic algorithm for large-scale transmit power control of dense and robust wireless networks in harsh industrial environments. *Applied Soft Computing*, 65:243–259, 2018.

[48] Raheleh Moallemi, Arash Bozorgchenani, and Daniele Tarchi. An evolutionary-based algorithm for smart-living applications placement in fog networks. In *2019 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2019.

[49] Carla Mouradian, Somayeh Kianpisheh, Mohammad Abu-Lebdeh, Fereshteh Ebrahimnezhad, Narjes Tahghigh Jahromi, and Roch H Glitho. Application component placement in nfv-based hybrid cloud/fog systems with mobile fog nodes. *IEEE Journal on Selected Areas in Communications*, 37(5):1130–1143, 2019.

[50] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. Fogplan: A lightweight qos-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, 6(3):5080–5096, 2019.

[51] Sudip Misra and Niloy Saha. Detour: Dynamic task offloading in software-defined fog for iot applications. *IEEE Journal on Selected Areas in Communications*, 37(5):1159–1166, 2019.

[52] Chu-ge Wu and Ling Wang. A deadline-aware estimation of distribution algorithm for resource scheduling in fog computing systems. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 660–666. IEEE, 2019.

[53] Ilias Gravalos, Prodromos Makris, Kostas Christodoulopoulos, and Emmanouel A Varvarigos. Efficient network planning for internet of things with qos constraints. *IEEE Internet of Things Journal*, 5(5):3823–3836, 2018.

[54] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized iot service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, 2017.

[55] Li Kuang, Tao Gong, Shuyin OuYang, Honghao Gao, and Shuiguang Deng. Offloading decision methods for multiple users with structured tasks in edge

computing for smart cities. *Future Generation Computer Systems*, 105:717–729, 2020.

[56] Om-Kolsoom Shahryari, Hossein Pedram, Vahid Khajehvand, and Mehdi Dehghan TakhtFooladi. Energy and task completion time trade-off for task offloading in fog-enabled iot networks. *Pervasive and Mobile Computing*, 74: 101395, 2021.

[57] Pedram Memari, Seyedeh Samira Mohammadi, Fariborz Jolai, and Reza Tavakkoli-Moghaddam. A latency-aware task scheduling algorithm for allocating virtual machines in a cost-effective and time-sensitive fog-cloud architecture. *The Journal of Supercomputing*, pages 1–30, 2021.

[58] Chu-ge Wu, Wei Li, Ling Wang, and Albert Y Zomaya. An evolutionary fuzzy scheduler for multi-objective resource allocation in fog computing. *Future Generation Computer Systems*, 117:498–509, 2021.

[59] Xin Gao, Xi Huang, Simeng Bian, Ziyu Shao, and Yang Yang. Pora: Predictive offloading and resource allocation in dynamic fog computing systems. *IEEE Internet of Things Journal*, 7(1):72–87, 2019.

[60] Xinchen Lyu, Wei Ni, Hui Tian, Ren Ping Liu, Xin Wang, Georgios B Giannakis, and Arogyaswami Paulraj. Optimal schedule of mobile edge computing for internet of things using partial information. *IEEE Journal on Selected Areas in Communications*, 35(11):2606–2615, 2017.

[61] Chun-Wei Tsai. Seira: An effective algorithm for iot resource allocation problem. *Computer Communications*, 119:156–166, 2018.

[62] Hani Sami and Azzam Mourad. Dynamic on-demand fog formation offering on-the-fly iot service deployment. *IEEE Transactions on Network and Service Management*, 17(2):1026–1039, 2020.

[63] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[64] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[65] Dabiah Ahmed Alboaneen, Huaglory Tianfield, and Yan Zhang. Glowworm swarm optimisation algorithm for virtual machine placement in cloud computing. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 808–814. IEEE, 2016.

[66] Filip De Turck. Efficient resource allocation through integer linear programming: a detailed example. *arXiv preprint arXiv:2009.13178*, 2020.

[67] Fengjunjie Pan, Jianjie Lin, Markus Rickert, and Alois Knoll. Resource allocation in software-defined vehicles: Ilp model formulation and solver evaluation. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 2577–2584. IEEE, 2022.

[68] Ashish Kaushal, Osama Almurshed, Areej Alabbas, Nitin Auluck, and Omer Rana. An edge-cloud infrastructure for weed detection in precision agriculture. In *2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 0269–0276, 2023. doi: 10.1109/DASC/PiCom/CBDCom/Cy59711.2023.10361387.

[69] Antonio Benitez-Hidalgo, Antonio J Nebro, Jose Garcia-Nieto, Izaskun Oregi, and Javier Del Ser. jmetalpy: A python framework for multi-objective optimization with metaheuristics. *Swarm and Evolutionary Computation*, 51:100598, 2019.

[70] Guido van Rossum. time - time access and conversions, 2023. URL `https://docs.python.org/3/library/time.html`.

[71] Guido van Rossum. timeit. URL `https://docs.python.org/3/library/timeit.html`.

[72] Giampaolo Rodia. psutil. URL `https://github.com/giampaolo/psutil`.

[73] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. Parsl:

Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, 2019.

[74] Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[75] Apache Software Foundation. Apache kafka. *Apache Software Foundation*, 2011. URL `https://kafka.apache.org/`.

[76] Andrew Tridgell and The Samba Team. Samba: A file and print server for unix. *USENIX Conference on File and Storage Technologies*, 1992. URL `https://www.samba.org/`.

[77] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

[78] Jerry Banks. *Discrete event system simulation*. Pearson Education India, 2005.

[79] Josip Zilic, Atakan Aral, and Ivona Brandic. Efpo: Energy efficient and failure predictive edge offloading. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 165–175, 2019.

[80] Jie Sun, Tianyu Wo, Xudong Liu, Rui Cheng, Xudong Mou, Xiaohui Guo, Haibin Cai, and Rajkumar Buyya. Cloudsimsfc: Simulating service function chains in multi-domain service networks. *Simulation Modelling Practice and Theory*, 120:102597, 2022.

[81] Horst Rinne. *The Weibull distribution: a handbook*. CRC press, 2008.

[82] Bao Pang, Yong Song, Chengjin Zhang, and Runtao Yang. Effect of random walk methods on searching efficiency in swarm robots for area exploration. *Applied Intelligence*, 51(7):5189–5199, 2021.

[83] Andreas F Molisch. *Wireless communications*, volume 34. John Wiley & Sons, 2012.

[84] FP Hwang, Shu-Jen Chen, and Ching-Lai Hwang. *Fuzzy multiple attribute decision making: Methods and applications*. Springer Berlin/Heidelberg, 1992.

[85] Milan Zeleny. Compromise programming. *Multiple criteria decision making*, 1973.

[86] David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[87] Absalom E Ezugwu, Verosha Pillay, Divyan Hirasen, Kershen Sivanarain, and Melvin Govender. A comparative study of meta-heuristic optimization algorithms for 0–1 knapsack problem: Some initial results. *IEEE Access*, 7: 43979–44001, 2019.

[88] Juan J Durillo, Antonio J Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize nsga-ii. In *2008 IEEE international symposium on parallel and distributed processing*, pages 1–8. IEEE, 2008.

[89] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2): 173–195, 2000.

[90] Alan Mathison Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society, Series 2*, 45:161–228, 1939.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[92] Dask Development Team. Dask: Library for dynamic task scheduling, 2016.

[93] SL Barr, S Johnson, X Ming, M Peppa, N Dong, Z Wen, C Robson, L Smith, P James, D Wilkinson, et al. Flood-prepared: A nowcasting system for real-time impact adaption to surface water flooding in cities. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 6:9–15, 2020.

[94] Jose Manuel Sanchez Vilchez, Imen Grida Ben Yahia, and Noël Crespi. Self-healing mechanisms for software defined networks. 2014.

[95] Srinivas Chippada, Clint N Dawson, Monica L Martínez, and Mary F Wheeler. A godunov-type finite volume method for the system of shallow water equations.

Computer methods in applied mechanics and engineering, 151(1-2):105–129, 1998.

[96] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[97] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[98] Fabrizio Balducci, Donato Impedovo, and Giuseppe Pirlo. Machine learning applications on agricultural datasets for smart farm enhancement. *Machines*, 6 (3):38, 2018.

[99] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. Federated learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(3):1–207, 2019.

[100] Ateyah Alzahrani, Ioan Petri, and Yacine Rezgui. Analysis and simulation of smart energy clusters and energy value chain for fish processing industries. *Energy Reports*, 6:534–540, 2020.

[101] Cristina Dimidov, Giuseppe Oriolo, and Vito Trianni. Random walks in swarm robotics: an experiment with kilobots. In *International conference on swarm intelligence*, pages 185–196. Springer, 2016.

[102] Ritika Lohiya and Ankit Thakkar. Application domains, evaluation data sets, and research challenges of iot: A systematic review. *IEEE Internet of Things Journal*, 8(11):8774–8798, 2020.

[103] Serverless faqs - amazon web services. URL `https://aws.amazon.com/serverless/faqs/`. Accessed: 2023-06-17.

[104] Technical requirements for kubernetes | apache openwhisk, . URL `https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/k8s-technical-requirements.md`. Accessed: 2023-06-17.

[105] Federated function as a service | funcx. URL `https://funcx.org/`. Accessed: 2023-06-17.

[106] Biljana L Risteska Stojkoska and Kire V Trivodaliev. A review of internet of things for smart home: Challenges and solutions. *Journal of cleaner production*, 140:1454–1464, 2017.

[107] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, 2000.

[108] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[109] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[110] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-based load management in federated distributed systems. In *NSDI*, volume 4, pages 15–15, 2004.

[111] Yanif Ahmad, Bradley Berg, Ugur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, et al. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884, 2005.

[112] Matthew Jankowski, Peter Pathirana, and Sean Allen. *Storm Applied: Strategies for real-time event processing*. Simon and Schuster, 2015.

[113] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, pages 239–250, 2015.

[114] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[115] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[116] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[117] Oozie - Apache Oozie Workflow Scheduler for Hadoop — oozie.apache.org. `https://oozie.apache.org/`, . [Accessed 22-Sep-2022].

[118] Azkaban — azkaban.github.io. `https://azkaban.github.io/`. [Accessed 22-Sep-2022].

[119] Apache Airflow. `https://airflow.apache.org/`, . [Accessed 22-Sep-2022].

[120] Node-RED — nodered.org. `https://nodered.org/`. [Accessed 22-Sep-2022].

[121] Openwhisk. `https://openwhisk.apache.org/`, . Accessed: April 25, 2023.

[122] OpenFaaS. Serverless functions made simple. `https://github.com/openfaas/faas`, 2016. Accessed: April 25, 2023.

[123] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[124] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[125] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(1):1–61, 2012.

[126] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[127] LIAAD. Yake!, 2023. URL `https://liaad.github.io/yake/`. [Online; accessed 17-June-2023].

[128] SkBlaz. Rakun, 2023. URL `https://github.com/SkBlaz/rakun`. [Online; accessed 17-June-2023].

[129] Mahmoud Gamal, Saber Jafarizadeh, Mehran Abolhasan, Justin Lipman, and Wei Ni. Mapping and scheduling for non-uniform arrival of virtual network function (vnf) requests. In *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, pages 1–6. IEEE, 2019.

[130] Hanine Tout, Azzam Mourad, Nadjia Kara, and Chamseddine Talhi. Multi-persona mobility: Joint cost-effective and resource-aware mobile-edge computation offloading. *IEEE/ACM Transactions on Networking*, 2021.

[131] Vinita Yadav, BV Natesha, and Ram Mohana Reddy Guddeti. Ga-pso: service allocation in fog computing environment using hybrid bio-inspired algorithm. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, pages 1280–1285. IEEE, 2019.

[132] Rajrup Ghosh and Yogesh Simmhan. Distributed scheduling of event analytics across edge and cloud. *ACM Transactions on Cyber-Physical Systems*, 2(4): 1–28, 2018.

[133] Huizhen Hao, Jie Zhang, and Qing Gu. Optimal iot service offloading with uncertainty in sdn-based mobile edge computing. *Mobile Networks and Applications*, pages 1–10, 2021.

[134] Masoumeh Ayoubi, Mohammadreza Ramezanpour, and Reihaneh Khorsand. An autonomous iot service placement methodology in fog computing. *Software: Practice and Experience*, 51(5):1097–1120, 2021.

[135] Kuljeet Kaur, Sahil Garg, Gagangeet Singh Aujla, Neeraj Kumar, Joel JPC Rodrigues, and Mohsen Guizani. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE communications magazine*, 56(2):44–51, 2018.

[136] Xingju Xie, Xiaojun Wu, and Qiao Hu. Bi-objective optimization for industrial robotics workflow resource allocation in an edge–cloud environment. *Applied Sciences*, 11(21):10066, 2021.

[137] Mahbuba Afrin, Jiong Jin, Ashfaqur Rahman, Yu-Chu Tian, and Ambarish Kulkarni. Multi-objective resource allocation for edge cloud based robotic workflow in smart factory. *Future Generation Computer Systems*, 97:119–130, 2019.

[138] Dadmehr Rahbari, Sabihe Kabirzadeh, and Mohsen Nickray. A security aware scheduling in fog computing by hyper heuristic algorithm. In *2017 3rd Iranian Conference on Intelligent Systems and Signal Processing (ICSPIS)*, pages 87–92. Ieee, 2017.

[139] Jiuhong Yu, Mengfei Wang, JH Yu, and Seyedeh Maryam Arefzadeh. A new approach for task managing in the fog-based medical cyber-physical systems using a hybrid algorithm. *Circuit World*, 2021.

[140] Xing Chen, Jianshan Zhang, Bing Lin, Zheyi Chen, Katinka Wolter, and Geyong Min. Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(3): 683–697, 2021.

[141] Guang Peng, Huaming Wu, Han Wu, and Katinka Wolter. Constrained multi-objective optimization for iot-enabled computation offloading in collaborative edge and cloud computing. *IEEE Internet of Things Journal*, 2021.

[142] Raafat O Aburukba, Taha Landolsi, and Dalia Omer. A heuristic scheduling approach for fog-cloud computing environment with stationary iot devices. *Journal of Network and Computer Applications*, 180:102994, 2021.

[143] BV Natesha and Ram Mohana Reddy Guddeti. Adopting elitism-based genetic algorithm for minimizing multi-objective problems of iot service placement in

fog computing environment. *Journal of Network and Computer Applications*, 178:102972, 2021.

[144] Raafat O Aburukba, Mazin AliKarrar, Taha Landolsi, and Khaled El-Fakih. Scheduling internet of things requests to minimize latency in hybrid fog–cloud computing. *Future Generation Computer Systems*, 111:539–551, 2020.

[145] Binh Minh Nguyen, Huynh Thi Thanh Binh, Bao Do Son, et al. Evolutionary algorithms to optimize task scheduling problem for the iot based bag-of-tasks application in cloud–fog computing environment. *Applied Sciences*, 9(9):1730, 2019.

[146] Yan Sun, Fuhong Lin, and Haitao Xu. Multi-objective optimization of resource scheduling in fog computing using an improved nsga-ii. *Wireless Personal Communications*, 102(2):1369–1385, 2018.

[147] Shun-shun Fang, Zheng-yi Chai, and Ya-lun Li. Dynamic multi-objective evolutionary algorithm for iot services. *Applied Intelligence*, 51(3):1177–1200, 2021.

[148] Sambit Kumar Mishra, Deepak Puthal, Joel JPC Rodrigues, Bibhudatta Sahoo, and Eryk Dutkiewicz. Sustainable service allocation using a metaheuristic technique in a fog server for industrial applications. *IEEE Transactions on Industrial Informatics*, 14(10):4497–4506, 2018.

[149] Juan Wang and Di Li. Task scheduling based on a hybrid heuristic algorithm for smart production line with fog computing. *Sensors*, 19(5):1023, 2019.

[150] Jiafu Wan, Baotong Chen, Shiyong Wang, Min Xia, Di Li, and Chengliang Liu. Fog computing for energy-aware load balancing and scheduling in smart factory. *IEEE Transactions on Industrial Informatics*, 14(10):4548–4556, 2018.

[151] Mohamed K Hussein and Mohamed H Mousa. Efficient task offloading for iot-based applications in fog computing using ant colony optimization. *IEEE Access*, 8:37191–37201, 2020.

[152] Nerijus Morkevicius, Algimantas Venčkauskas, Nerijus Šatkauskas, and Jevgenijus Toldinas. Method for dynamic service orchestration in fog computing. *Electronics*, 10(15):1796, 2021.

[153] Tanissia Djemai, Patricia Stolf, Thierry Monteil, and Jean-Marc Pierson. A discrete particle swarm optimization approach for energy-efficient iot services placement over fog infrastructures. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 32–40. IEEE, 2019.

[154] Reinout Eyckerman, Siegfried Mercelis, Johann Marquez-Barja, and Peter Hellinckx. Requirements for distributed task placement in the fog. *Internet of Things*, 12:100237, 2020.

[155] Jianhua Fan, Xianglin Wei, Tongxiang Wang, Tian Lan, and Suresh Subramaniam. Deadline-aware task scheduling in a tiered iot infrastructure. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–7. IEEE, 2017.

[156] Muhammad Babar Kamal, Nadeem Javaid, Syed Aon Ali Naqvi, Hanan Butt, Talha Saif, and Muhammad Daud Kamal. Heuristic min-conflicts optimizing technique for load balancing on fog computing. In *International Conference on Intelligent Networking and Collaborative Systems*, pages 207–219. Springer, 2018.

[157] Ashwin Ashok, Peter Steenkiste, and Fan Bai. Vehicular cloud computing through dynamic computation offloading. *Computer Communications*, 120: 125–137, 2018.

[158] Mustafa I Khaleel and Michelle M Zhu. Adaptive virtual machine migration based on performance-to-power ratio in fog-enabled cloud data centers. *The Journal of Supercomputing*, pages 1–40, 2021.

[159] Qing Zhang, Xiaoyong Lin, Yongsheng Hao, and Jie Cao. Energy-aware scheduling in edge computing based on energy internet. *IEEE Access*, 8: 229052–229065, 2020.

[160] Lyla Naghipour Vijouyeh, Masoud Sabaei, José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Efficient application deployment in fog-enabled infrastructures. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2020.

[161] Mohammed Islam Naas, Laurent Lemarchand, Jalil Boukhobza, and Philippe Raipin. A graph partitioning-based heuristic for runtime iot data placement

strategies in a fog infrastructure. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 767–774, 2018.

[162] Lin Gu, Deze Zeng, Song Guo, Ahmed Barnawi, and Yong Xiang. Cost efficient resource management in fog computing supported medical cyber-physical system. *IEEE Transactions on Emerging Topics in Computing*, 5(1):108–119, 2015.

[163] Randa M Abdelmoneem, Abderrahim Benslimane, and Eman Shaaban. Mobility-aware task scheduling in cloud-fog iot-based healthcare architectures. *Computer Networks*, 179:107348, 2020.

[164] Amira Rayane Benamer, Hana Teyeb, and Nejib Ben Hadj-Alouane. Penguin search aware proactive application placement. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 229–244. Springer, 2019.

[165] Saeed Javanmardi, Mohammad Shojafar, Valerio Persico, and Antonio Pescapè. Fpfts: a joint fuzzy particle swarm optimization mobility-aware approach to fog task scheduling algorithm for internet of things devices. *Software: Practice and Experience*, 51(12):2519–2539, 2021.

[166] Ramyad Hadidi, Jiashen Cao, Michael S Ryoo, and Hyesoon Kim. Toward collaborative inferencing of deep neural networks on internet-of-things devices. *IEEE Internet of Things Journal*, 7(6):4950–4960, 2020.

[167] Mohammed Islam Naas, Philippe Raipin Parvedy, Jalil Boukhobza, and Laurent Lemarchand. ifogstor: an iot data placement strategy for fog infrastructure. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 97–104. IEEE, 2017.

[168] Antonino Rullo, Edoardo Serra, Elisa Bertino, and Jorge Lobo. Shortfall-based optimal placement of security resources for mobile iot scenarios. In *European Symposium on Research in Computer Security*, pages 419–436. Springer, 2017.

[169] Xuan-Qui Pham and Eui-Nam Huh. Towards task scheduling in a cloud-fog computing system. In *2016 18th Asia-Pacific network operations and management symposium (APNOMS)*, pages 1–4. IEEE, 2016.

[170] Simone Bolettieri, Raffaele Bruno, and Enzo Mingozzi. Application-aware resource allocation and data management for mec-assisted iot service providers. *Journal of Network and Computer Applications*, 181:103020, 2021.

[171] Abbas Najafizadeh, Afshin Salajegheh, Amir Masoud Rahmani, and Amir Sahafi. Multi-objective task scheduling in cloud-fog computing using goal programming approach. *Cluster Computing*, pages 1–25, 2021.

[172] Mohamed Abdel-Basset, Reda Mohamed, Mohamed Elhoseny, Ali Kashif Bashir, Alireza Jolfaei, and Neeraj Kumar. Energy-aware marine predators algorithm for task scheduling in iot-based fog computing applications. *IEEE Transactions on Industrial Informatics*, 17(7):5068–5076, 2020.

[173] John Paul Martin, A Kandasamy, and K Chandrasekaran. Crew: Cost and reliability aware eagle-whale optimiser for service placement in fog. *Software: Practice and Experience*, 50(12):2337–2360, 2020.

[174] Liang Zhao. Privacy-preserving distributed analytics in fog-enabled iot systems. *Sensors*, 20(21):6153, 2020.

[175] Kai Lin, Sameer Pankaj, and Di Wang. Task offloading and resource allocation for edge-of-things computing on smart healthcare systems. *Computers & Electrical Engineering*, 72:348–360, 2018.