A Dynamic Distributed Scheduler for Computing on the Edge

by

Fei Hu

B.A., Jilin University, 2016

M.S., University of Electronic Science and Technology of China, 2019

M.S., University of Colorado Boulder, 2023

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2024

Committee Members:

Shivakant Mishra, Chair

Qin Lv

Lijun Chen

Mohammad AlMutawa

Nuha Albadi

Hu, Fei (Ph.D., Computer Science)

A Dynamic Distributed Scheduler for Computing on the Edge

Thesis directed by Prof. Shivakant Mishra

Edge computing is vital for IoT applications that require rapid, secure data processing, yet these applications are resource-heavy, and edge resources are often outmatched by cloud capabilities. Efficient resource utilization is crucial to fulfill application demands like latency, privacy, and cost. Static scheduling systems frequently fail to meet these needs in dynamic, hybrid IoT environments. This thesis details the design, implementation, and testing of a dynamic distributed scheduler tailored for edge environments. It orchestrates task execution across all available edge resources to meet the diverse requirements of IoT applications, including lightweight AI applications running in containers and resource-intensive DNN applications. Central to this scheduler is its ability to continuously monitor and adapt to the changing state of the IoT infrastructure, adjusting task scheduling based on real-time environmental and system conditions. The practicality of this scheduler is confirmed through a prototype that has been evaluated across various AI applications. To enhance the architecture further, various middleware layers have been evaluated to deepen our understanding and establish a foundation for future improvements. The research pioneers an efficient AI system designed to handle a variety of user requests at the edge.

This collaborative research effort extends previous studies, recognizing the contributions of fellow researchers and the wider academic community. We are thankful for the support provided by NSF grants.

Dedication

This thesis is dedicated to my family.

Acknowledgements

I owe a profound debt of gratitude to my advisor, Professor Shivakant Mishra, whose steadfast support, invaluable guidance, and insightful feedback have been crucial throughout my doctoral studies. His mentorship has not only shaped my research direction but has also significantly contributed to my personal and academic development.

I am also thankful to my dissertation committee members, Professor Mohammad AlMutawa, Professor Qin Lv, and Professor Lijun Chen, for their expert advice and constructive feedback which have greatly enhanced the quality of my research.

Special thanks go to my labmates, Maram Kurdi and Nuha Albadi, whose commitment to their work and life perspectives have greatly influenced my own philosophy and approach to life's challenges. I am also grateful to Ji's family, Xiaoyuan's family, and Yifeng for their unwavering support and companionship, which have been a source of joy and comfort throughout my PhD journey.

My deepest appreciation extends to my family, whose love, encouragement, and sacrifices have been the foundation of my entire educational pursuit. To my best friend and soulmate, Jinpeng, your constant support has been essential in my journey towards this PhD. I am immensely grateful for your love. To my children, Xander and Layla, who are the greatest gifts of my life, you remain the core of my heart and my endless motivation.

Lastly, I acknowledge myself, Fei, for persevering through challenges. This journey of traversing mountains and rivers is a testament to endurance, and it reassures that time will eventually showcase the efforts spent.

Contents

Chapter

1 Introduction **1**

2 Background **4**

2.1 Background . . . 4

2.1.1 Cloud Computing . . . 4

2.1.2 Edge Computing . . . 5

2.1.3 Applications on the Edge . . . 9

2.2 Related Work . . . 10

2.2.1 Scheduling on the Edge . . . 10

2.2.2 DNN Partitioning . . . 12

2.2.3 Middleware Layer . . . 14

3 Pre-Work **16**

3.1 Device Profile Evaluation . . . 16

3.1.1 Face Detection Container . . . 17

3.1.2 Profile Evaluation Scenarios . . . 18

3.1.3 Profile Evaluation Results . . . 18

3.2 AI Applications Using Distributed Systems . . . 20

3.2.1 Methodology . . . 20

3.2.2 Experiment Results . . . 22

3.2.3 Best Scenario . . . 24

4 Dynamic Distributed Scheduler **28**

4.1 Scheduler Architecture . . . 28

4.1.1 Scheduler Design Principles . . . 29

4.1.2 Device Profiling . . . 30

4.1.3 Predictive Profiling . . . 32

4.1.4 Architecture Components . . . 32

4.1.5 Architecture Workflow . . . 34

4.2 Experiment with an AI Application . . . 36

4.2.1 An AI Application . . . 36

4.2.2 Results and Analysis . . . 37

4.3 Experiment with an AR Application . . . 43

4.3.1 An Augmented Reality Application . . . 43

4.3.2 Set Up . . . 44

4.3.3 Results and Analysis . . . 44

4.3.4 Analysis . . . 47

4.4 Optimization . . . 48

4.4.1 Predictive Profiling and Dynamic Update . . . 48

4.4.2 Balance Between the Load on the Edge Server and Success Rate . . . 49

4.5 Conclusion . . . 53

5 A Dynamic Distributed Scheduler for DNN Inference on the Edge **55**

5.1 Background . . . 55

5.2 Related Work . . . 56

5.3 Scheduler Design . . . 58

5.3.1 Scheduler Design Principles . . . 59

5.3.2 Device Profiling . . . 60

5.3.3 Architecture Workflow 60
5.4 DNN Model Evaluation 62
5.4.1 Characterizing Layers in DNN Models 62
5.4.2 Pre-evaluation of DNN Model 64
5.5 Evaluation 66
5.5.1 Case1: Raspberry Pi and Edge Server 66
5.5.2 Case2.1: Rasp and GPU 67
5.5.3 Case2.2: Rasp, GPU, and Edge Server 67
5.6 Conclusion 68

6 Middleware Evaluation **70**
6.1 IoT Architectures and Scenarios 72
6.1.1 Middleware Architectures 72
6.1.2 Edge-IoT Scenarios 75
6.2 Middleware Evaluation 78
6.2.1 Experiment Setup 78
6.2.2 Qualitative Evaluation 80
6.2.3 Performance Measurements 80
6.3 Discussion 87
6.4 Conclusion 90

7 Conclusion **91**
7.1 Research Summary 91
7.2 Future Work 92

Bibliography **94**

List of Tables

Table

3.1 System Configuration . . . 17

3.2 Runtime for Different Image Size . . . 17

3.3 Profile of Cold Container on the Edge Server . . . 18

3.4 Profile of Cold Container on the Raspberry Pi . . . 19

3.5 Profile of Warm Container on the Edge Server . . . 19

3.6 Profile of Warm Container on the Raspberry Pi . . . 20

3.7 System Configuration of Distributed AR Application . . . 22

4.1 Devices Profile with 1080p Video . . . 44

4.2 Predictive Profile . . . 48

4.3 Adjusted Device Profile . . . 51

5.1 Literature review on DNN Partition . . . 57

5.2 DNN System Configuration . . . 61

5.3 Output Data Size at Each Layer . . . 64

5.4 Sub-Model Latency . . . 65

5.5 Success Rate . . . 66

5.6 Edge Device Utilization Rate . . . 67

6.1 Fully-automated Scenarios average end-to-end latency . . . 80

6.2 Average Throughput for Fully-automated Scenarios . . . 82

6.3 Average Latency for Human-in-the-loop Scenarios D and E 83

6.4 Average Latency for Human-in-loop Scenario F 84

List of Figures

Figure

2.1 Edge Computing Scenario 5
2.2 Edge Server 7
2.3 Collaborative Computing 8

3.1 An Example of Face Detection 16
3.2 Distributed System Architecture 20
3.3 All On the Edge Server Architecture 23
3.4 All on the Edge Server Results 23
3.5 Distribute Across Computing Nodes Architecture 25
3.6 Distribute Across Computing Nodes Results 25
3.7 Best Output scenario 26

4.1 Architecture 32
4.2 Experiment Application 37
4.3 50 Images 40
4.4 1000 Images 41
4.5 Relationship between CPU Load and performance 41
4.6 DDS vs DDSwithR2 43
4.7 Experiment Results with 240p Video 45
4.8 Experiment Results with 480p Video 45

4.9 Experiment Results with 1080p Video . . . 47
4.10 End-to-End Time . . . 47
4.11 Predictive profiling vs Actual Profiling . . . 48
4.12 Dynamic Scheduler Algorithm . . . 49
4.13 Extend Dynamic Scheduler Algorithm . . . 49
4.14 10 Percentage Increase . . . 51
4.15 30 Percentage Increase . . . 52
4.16 50 Percentage Increase . . . 53

5.1 A 5-layer DNN Classifies Inference . . . 58
5.2 A DNN Inference Distribution Example . . . 59
5.3 VGG16 Architecture . . . 63
5.4 Success Rate with Variable GPU Load . . . 68

6.1 Baseline Architecture . . . 73
6.2 Edgex Based Middleware Architecture . . . 74
6.3 Kura Based Middleware Architecture . . . 75
6.4 End-to-end latency for fully automated scenarios . . . 81
6.5 Average Throughput for Fully-Automated Scenarios . . . 83
6.6 Average Latency for Human-in-the-Loop Scenarios D and E . . . 84
6.7 Average Latency for Human-in-the-Loop Scenario F . . . 85
6.8 Average Latency for Scalability Scenario G . . . 86
6.9 Average Latency for Scalability Scenario H . . . 86

Chapter 1

Introduction

Edge AI systems [1, 2] have emerged as a promising solution for handling the growing demand for AI workloads by bringing computation closer to the data sources and reducing latency [3, 4]. A key challenge in building AI applications at the edge is that these applications tend to be highly compute-intensive and computing capabilities are limited at the edge server when compared to the cloud. Developers have been addressing this issue in an ad hoc manner by distributing different computing tasks of applications across different computing nodes available at the edge, including different low-end sensing devices as well as different processing units at the edge server. However, distributing computing tasks across different computing nodes comes with the added cost of having to transfer data between nodes. Thus, a careful balance between the performance improvement due to parallel computation across different computing nodes and the increased delays due to data movement is needed to fully realize the performance benefits of task distribution. This is further complicated by three important factors prevalent in an IoT environment: (1) network conditions change dynamically resulting in communication delays of data movement changing dynamically; (2) CPU load on computing nodes changes dynamically due to dynamically varying contexts resulting in compute times of various tasks changing dynamically; (3) compute nodes at the edge are highly diverse with widely varying compute, storage, and communication capabilities [3, 5, 6].

This thesis addresses these challenges by building a dynamic distributed scheduler that schedules different computing tasks of an application across different computing nodes based on the current computing and networking conditions to meet all the requirements of the application, in-

cluding latency, privacy, power, and cost constraints. This scheduler constantly monitors the current compute and network conditions and dynamically adjusts the scheduling of the compute tasks accordingly. It has five important features:

(1) Device profiling: To address heterogeneity among different computing devices, each device in the IoT environment is profiled for different computing tasks under different computing loads to enable the scheduler to make optimal scheduling decisions at runtime.

(2) Predictive profiling: To address the limitation of pre-profiling every possible device under every possible computing environment, the scheduler incorporates a prediction mechanism to predict the running times of various computing tasks during runtime and adjusts schedules accordingly.

(3) Two-level scheduling decision: To minimize the overhead of the scheduler, the scheduling decision is done at two levels, a primitive scheduling decision at the low-end devices and a near-optimal scheduling decision at the edge server.

(4) Local compute principle: To address the uncertainty in determining accurate communication latency under every possible computing environment, the scheduler schedules tasks locally as a low-level scheduling decision as long as this scheduling satisfies all the application requirements.

(5) Predictive compute drops: The scheduler selectively drops some compute task instances based on application guidance to maximize meeting the overall application constraints.

For resource-intensive DNN applications, edge devices, constrained by limited computing capabilities, may struggle to independently handle the entire workload. Contemporary research mitigates this by segmenting DNN models into sub-models, thus allowing edge devices to collaboratively execute a single inference task via parallel computing. This thesis concentrates on the challenge of effectively partitioning DNN models and outlines a set of strategies to address this.

Building on the work outlined above, this thesis establishes an architecture that supports a scheduling system capable of efficiently managing variable user requests including light-weight AI applications and deep learning applications at the edge. At present, there is a large gap between the IoT application developer requirements and the underlying IoT infrastructures [7]. IoT application developers require advanced system support services like resource discovery, sharing of IoT devices, and robust computing and communication abstractions for developing context-aware applications [8] [9]. However, existing IoT devices and software infrastructures are often developed in isolation, lacking portability and code reusability. An edge computing middleware layer aims to address these issues. This thesis assesses prominent middleware layers to understand the abstractions they offer, their applicability across various IoT applications, their ease of use, and their impact on application performance. In particular, we evaluate two popular edge computing middleware layers, EdgeX Foundry and Kura.

This thesis is organized into six chapters. Chapter 1 provides an overview of the entire study. Chapter 2 delves into the background and reviews related literature. Chapter 3 focuses on preliminary work, encompassing device profiling and the role of AR applications in distributed systems. Chapter 4 presents the principles underlying the Dynamic Distributed Scheduler, along with experimental results across various applications and scenarios. Chapter 5 expands on the dynamic distributed scheduler to address DNN inference at the edge, while Chapter 6 delves into middleware evaluation and provides a thorough analysis.

Chapter 2

Background

2.1 Background

2.1.1 Cloud Computing

The advancement of processing and storage technologies, coupled with the widespread success of the Internet, has made computing resources more affordable, powerful, and accessible than ever. This progress has paved the way for the emergence of cloud computing—a model where computing resources like CPU and storage are offered as utilities over the Internet, available on demand. In this model, the role of service providers splits into infrastructure providers, who manage cloud platforms and offer resources on a pay-per-use basis, and service providers, who utilize these resources to deliver services to end-users [10, 11].

Cloud computing presents several compelling advantages, such as eliminating the need for upfront investment due to its pay-as-you-go pricing model. This allows service providers to rent necessary resources based on their current needs and pay only for what they use, effectively removing the barrier to entry. Operating costs are significantly reduced since resources can be rapidly scaled up or down in response to demand, avoiding the necessity to provision for peak loads and allowing for cost savings during low-demand periods. The scalability feature of cloud computing, sometimes referred to as surge computing, enables service providers to easily expand their services to accommodate rapid increases in demand. Furthermore, cloud-hosted services are web-based and thus easily accessible from a variety of internet-connected devices, including not only computers but

also smartphones and PDAs. By outsourcing service infrastructure to the cloud, service providers can shift business risks, such as hardware failures, to infrastructure providers who possess more expertise in managing these issues. [12, 13].

However, this evolution also brings forth new challenges, particularly in terms of security, privacy, and data governance. As reliance on cloud services grows, so does the importance of implementing robust cybersecurity measures and ensuring compliance with data protection regulations. The shared responsibility model of cloud computing necessitates a proactive approach to security, with both providers and users playing critical roles in safeguarding data. [10, 11, 12, 13].

2.1.2 Edge Computing

Edge computing extends centralized applications and data to the logical extremity of the network, bringing them closer to the data source. This proximity reduces latency and decreases data transfer costs. Specifically, end devices, edge servers, and computing across edge devices can all serve as computing resources, provided they effectively handle computing tasks. [1, 2, 3, 4]

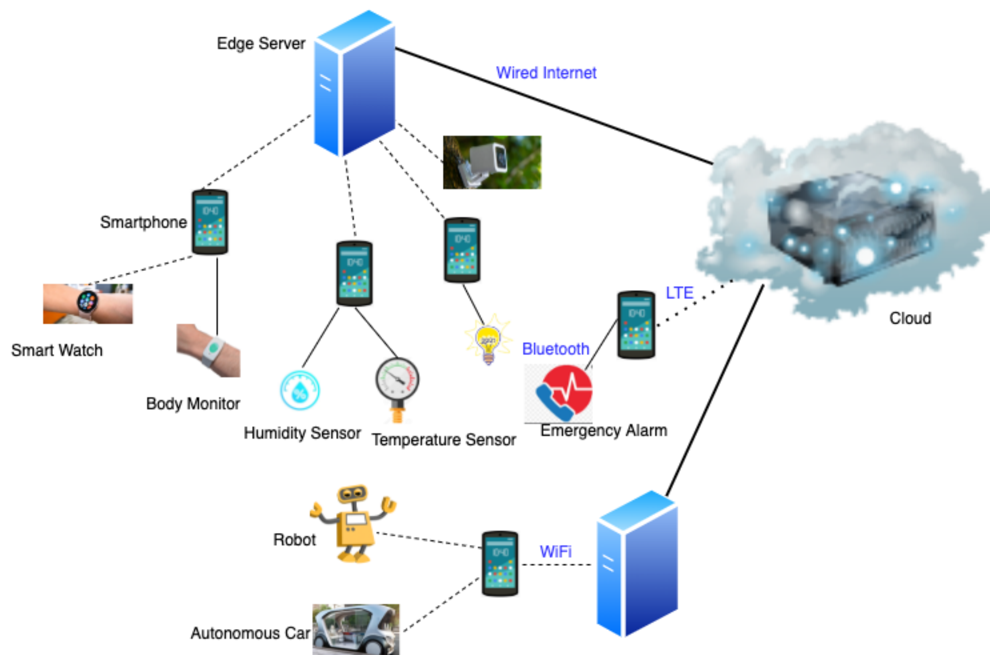


Figure 2.1: Edge Computing Scenario

Figure 2.1 illustrates a typical edge computing scenario, which encompasses the cloud, edge servers, and a variety of distributed devices. These devices range from stationary units, such as cameras, emergency alarms, and environmental sensors, to mobile elements, including autonomous vehicles, smartphones, smartwatches, and health monitors. End devices establish connections to nearby edge servers using WiFi or Bluetooth, while edge servers link to the cloud via wired internet. Smartphones may also use LTE as an additional connectivity option. These end devices generate streaming data and exhibit a wide diversity in computing power, storage capacity, and communication capabilities. For instance, autonomous vehicles, smartphones, and robots can process part of this data locally. Autonomous vehicles make real-time driving decisions, a crucial feature for responding swiftly in emergencies. Similarly, robots act based on local computations. Given that end devices typically possess limited computing resources, some data must be forwarded to edge servers for more complex processing. In local networks, autonomous vehicles can sync data with edge servers to obtain a comprehensive view of the local environment, aiding in better decision-making. When edge servers are overwhelmed, the cloud serves as a fallback, although reliance on distant cloud servers can result in higher latency and increased data transfer costs.

In the described scenario, determining the appropriate node for task processing should be based on the task's characteristics and the computational resources of the node. To efficiently offload tasks to edge devices, an integrated framework is required—one that can identify available resources in real-time, support scalable resource sharing, facilitate communication between end nodes, and incorporate optimal scheduling algorithms. Currently, there is no framework available that offers such advanced and intelligent computation capabilities. Therefore, my dissertation will concentrate on designing and implementing an integrated edge AI microservice-based infrastructure. This infrastructure aims to deliver services to users while optimizing for resource efficiency, scalability, latency, security, and other critical factors.

2.1.2.1 On-Device Computing

Some end devices, like smartphones, generate data but have limited computing power, allowing them to function as edge servers under certain conditions. The primary challenge lies in their restricted computing resources, necessitating that any deployed deep learning model be as compact as possible while still ensuring quality of service (QoS). To address this, one approach involves utilizing models designed for devices with constrained resources, such as MobileNet [14] and SqueezeNet [15]. Alternatively, compressing deep learning models by methods such as pruning connections between neurons or removing neurons altogether can reduce the number of parameters, saving space and computational resources. However, this compression can impact model performance, making it crucial to find an optimal balance between resource use and model effectiveness[16, 3, 4].

2.1.2.2 Edge Server Computing

Edge servers, possessing significantly more computing resources than end devices yet substantially less than cloud servers, represent a middle ground in computational capacity. The distinctions and relationships among these components are illustrated in Figure 2.2.

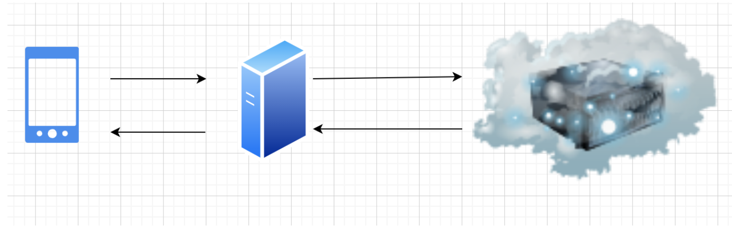


Figure 2.2: Edge Server

Cloud computing often faces challenges with latency, scalability, and privacy. In contrast, edge servers reduce end-to-end latency by being closer to users and typically employ a hierarchical architecture to overcome scalability issues. By processing data locally, edge servers also mitigate privacy risks associated with transmitting personal information to the cloud [10, 11].

Once trained in the cloud, deep learning models can be deployed on edge servers for local data processing. This approach, referred to as an offline model, allows for immediate data analysis

without the need for continuous cloud connectivity. As demonstrated in studies like the one by Wang et al.[17] deploying an offloaded DNN model on edge servers is a common practice. Moreover, when resources permit, edge servers can further refine these models with newly acquired data, enhancing their performance over time.

A critical aspect of managing edge servers is the efficient scheduling of computing resources to maximize their utilization. Optimizing this process ensures that edge servers can effectively handle the computational demands placed upon them while providing timely and secure data processing services.

2.1.2.3 Computing Across Edge Devices

Leveraging edge computing fully often involves distributed computing across edge devices, which can be categorized into several offloading scenarios: binary offloading, partial offloading, hierarchical architectures, and distributed computing.

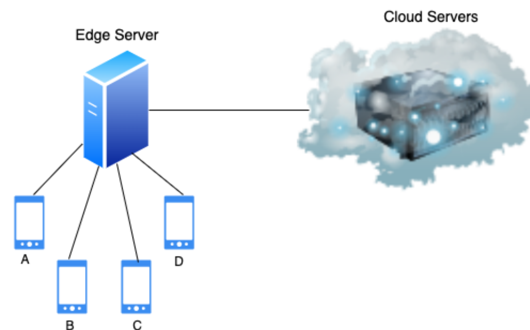


Figure 2.3: Collaborative Computing

Binary offloading refers to the decision of whether a deep learning model operates entirely offline. Partial offloading, on the other hand, involves offloading only segments of the model for processing. This report specifically aims to delve deeper into hierarchical architectures and distributed computing.

In hierarchical architectures, offloading occurs across a spectrum of devices, from end devices to edge servers and cloud servers. Cloud computing excels in processing capacity, making it suitable

for heavy computational tasks, while edge computing offers rapid processing by being in closer proximity to the data source. However, each approach has its limitations. Sending all data to the cloud can consume excessive time and bandwidth, whereas relying solely on edge nodes, which may lack sufficient energy and resources, can result in processing failures. Finding a method to combine these two computing paradigms could potentially offset their drawbacks and harness their strengths more effectively. This thesis will introduce more about the state-of-art of computing across edge devices in the next section.

2.1.3 Applications on the Edge

As smartphones and IoT sensors become more prevalent, the volume of data requiring real-time processing has surged, making deep learning an essential tool for analysis and decision-making. Below are several applications where deep learning is currently being extensively utilized [16].

2.1.3.1 Computer Vision

Computer vision focuses on enabling computers to derive high-level understanding from digital images or videos, mimicking the processing capabilities of the human visual system. This field encompasses technologies for acquiring, processing, and analyzing data from video sequences or other image-capturing devices. Key sub-domains include scene reconstruction, video tracking, and object recognition, among others. As these areas have evolved, numerous successful deep-learning models have been developed and widely adopted. For instance, systems like Vigil are employed to manage traffic and congestion by leveraging computer vision techniques [18].

2.1.3.2 Natural Language Processing

Natural Language Processing (NLP) facilitates interactions between computers and humans, tracing its origins back to the 1950s. However, it is the advancements in deep learning that have truly unlocked its potential, accelerating technological progress. NLP encompasses a wide range of tasks, including text and speech processing, morphological analysis, lexical semantics, and more

sophisticated applications. Products like Amazon Alexa and Apple Siri exemplify NLP's application at the edge, demonstrating its practicality in everyday technologies [19].

2.1.3.3 Internet of Things

oT sensors play a pivotal role in the development of smart cities, with devices such as wearable sensors and health detection sensors generating vast and varied data types. Processing these multiple data streams represents a significant area of exploration. Specifically, IoT devices, compared to servers endowed with ample computing resources, often have limited computational capacity. Therefore, the challenge of compressing deep learning models to accommodate these less powerful devices is a topic of widespread interest. Further details on this subject will be discussed in the section on Edge Computing [20, 21].

2.1.3.4 Virtual Reality and Augmented Reality

Virtual Reality (VR) creates a simulated environment that offers an immersive experience, making users feel as if they are truly part of it [22]. VR has found broad applications in education, healthcare, and entertainment, particularly in video and 3D gaming. Other forms of virtual reality, such as Augmented Reality (AR) and Extended Reality (XR), expand on this concept. AR, for instance, enhances the interactive experience by overlaying digital information on real-world objects, enriching the user's perception. It's extensively used in urban design, visual arts, and architecture, demonstrating significant potential in emergency rescue, health planning, and other fields [23, 24].

2.2 Related Work

2.2.1 Scheduling on the Edge

In edge computing, task scheduling confronts several pivotal challenges [3, 4]. Resource constraints mean that edge devices often lack the computational power, memory, and storage capabilities of the cloud servers, complicating the efficient handling of complex or resource-heavy tasks. The heterogeneous nature of devices—each with distinct computational capacities, operating systems,

and networking technologies—adds complexity to task allocation. Moreover, network fluctuations can lead to inconsistent connectivity and variable bandwidth, disrupting timely task coordination [5, 25]. The network's dynamic character, particularly with mobile devices, necessitates that schedulers dynamically adapt to fluctuations in network topology and resource availability [6]. Ensuring privacy and security is paramount, given the processing of sensitive data near its origin requires rigorous data protection protocols [26]. Energy efficiency remains crucial, especially for battery-operated devices, to prolong device lifespan and minimize operational costs [27]. Furthermore, reducing latency is essential in applications demanding real-time or near-instant processing, urging schedulers to finely tune task execution and data transmission to meet stringent latency requirements.

To overcome the above challenges, in recent years, many state-of-the-art scheduling techniques have emerged and those efforts primarily fall into two categories: centralized scheduling and distributed scheduling [28]. Generally, centralized methods mainly include convex optimization [29, 30, 31, 32, 33], approximate algorithm [34, 35**?** , 36], heuristic algorithm [37, 38, 39, 40, 41], and machine learning [42, 43, 44, 45, 46]; distributed methods mainly include game theory [47, 48, 49, 50, 51], matching theory [52, 53, 54, 55, 56], auction [57, 58, 59, 60, 61], federated learning [62, 63, 64], and blockchain [65, 66, 67, 68, 69].

The referenced studies highlight performance improvements in areas like latency, energy use, cost, utility, profit, and resource efficiency. For example, paper [29] leverages Lyapunov optimization to enhance problem-solving efficiency, with a focus on minimizing response times. Meanwhile, paper [38] addresses the distribution challenge as NP-hard, proposing a greedy heuristic algorithm aimed at satisfying latency requirements. Additionally, paper [48] targets energy efficiency with a distributed algorithm derived from Game Theory.

While these studies present significant benefits in performance metrics, their real-world application faces notable challenges. Techniques like Lyapunov optimization and decomposition in complex algorithms may overly complicate implementation in actual edge networks [29]. Similarly, approaches based on approximation algorithms, game theory, and auctions risk converging to local optima [34, 47, 57]. Heuristic and machine learning algorithms often require extensive parameter-

ization, complicating their adaptability [37, 42]. Matching theory, although considered, may not effectively address partial offloading issues, which are central to this paper [52, 53]. Moreover, emerging methods like federated learning and blockchain, while promising, are still nascent in this domain [62, 65].

Bernstein discusses container technology's evolution from Unix to Docker and Kubernetes, underscoring their role in enhancing cloud-based application deployment and management. [70]. Paper [71] assesses Docker based on deployment and termination, resource and service management, fault tolerance, and caching. This evaluation highlights Docker's potential to enhance the efficiency of EC infrastructures for improved application response times and user experience at the network's edge, making it a promising platform for EC. The paper [72] investigates how container technologies optimize the virtualization of IoT edge devices, specifically evaluating the performance effects of implementing container-based virtualization in edge computing scenarios. Along with other research [73, 74], the above studies highlight the extensive adoption of containers in edge network environments.

We introduce a dynamic distributed scheduler that utilizes device profiling and workload monitoring to address the mentioned challenges. Employing a two-tier scheduling approach, the lower level emphasizes local computing, with comprehensive performance enhancements elucidated in this paper. Additionally, this framework sets the stage for upper-level scheduling, where we can delve into performance enhancement based on current research findings.

2.2.2 DNN Partitioning

A fractional offloading strategy, which takes advantage of the distinctive layered structure of Deep Neural Networks (DNNs), can be employed. This strategy involves dividing the DNN model so that certain layers are processed on the device itself, while others are handled by the edge server or cloud. This technique, known as DNN model partitioning, has the potential to decrease latency by utilizing the computational power of additional edge devices [16].

The process for identifying the optimal partition point involves three key steps: first, assessing

and modeling the resource consumption of various DNN layers and the volume of intermediate data exchanged between them; second, estimating the total operational cost based on the configurations of specific layers and available network bandwidth; and third, selecting the most suitable partition point from among the candidates, taking into consideration factors such as delay and energy requirements [75, 76].

An alternative approach to model segmentation involves vertical partitioning, specifically for Convolutional Neural Networks (CNNs). Unlike horizontal partitioning, which splits the model layer by layer, vertical partitioning segments the layers into a grid pattern. This method effectively divides the CNN layers into separate computation tasks that can be distributed independently [77].

Horizontal partitioning of the Deep Learning (DL) model, distributing it across the end, edge, and cloud layers, is the most prevalent method of segmentation. The difficulty rests in smartly determining where to partition the DNN model. COMET offloads a thread solely based on whether its execution time surpasses a predefined threshold, disregarding other factors like data transfer volume or wireless network availability [78]. Odessa, on the other hand, bases its computation partitioning decisions on the execution time and data requirements of only a portion of the function, neglecting the broader context of the entire application [79]. CloneCloud applies the same offloading criteria across all instances of a given function [80]. In contrast, MAUI represents an improvement in offloading decision-making by evaluating each function invocation individually and taking the entire application into account when determining which functions to offload [81]. Paper [82] introduces an on-demand, low-latency inference framework that optimizes both the model partition strategy, tailored to the disparate computational capacities of mobile devices and edge servers, and the early exit strategy, adapted to complex network conditions. Various strategies have been devised to distribute a pre-trained DNN across multiple mobile devices, aiming to speed up DNN inference on these devices [83, 84, 85, 86]. Bhardwaj et al. expanded on this by incorporating memory and communication costs into the distributed inference framework. To tackle these challenges, they suggested model compression techniques and a network science-based algorithm for partitioning knowledge [87, 88].

2.2.3 Middleware Layer

At the edge, a vast array of devices, each running different operating systems and software, connect to the internet using various protocols. This diversity creates a challenge in integrating these heterogeneous applications seamlessly. Edge middleware emerges as a solution to bridge this gap, enabling the cohesive functioning of diverse applications [89, 90, 91, 92].

Key challenges in developing IoT middleware, as identified in the literature, encompass heterogeneity, mobility, security, and privacy. The wide variety of IoT devices and their software complicates achieving interoperability. The mobility of some IoT devices leads to a constantly changing network topology, affecting network parameters and causing potential actuation conflicts. Moreover, ensuring the security and privacy of data generated by these devices, as well as the personal data of users, is paramount, especially in an open computing environment where these concerns are magnified [89, 90, 92, 93, 94, 95].

In recent years, a number of edge middleware systems have been introduced, with some being open-source platforms like EdgeX Foundry [96], Azure IoT Edge [97], and Kura [98], while others are proprietary solutions, including Zededa [99] and Linux Foundation Edge (LF Edge)[100]. Furthermore, the research community has proposed various middleware layers to address specific needs. These contributions include a middleware layer designed for cost-efficient and flexible multi-tenant applications, as well as a middleware layer tailored for mobile terminals, showcasing the ongoing efforts to enhance middleware technology for diverse applications and environments [101, 102].

LF Edge [100] serves as an umbrella organization with the goal of creating an open and interoperable framework for edge computing, which is independent of hardware, silicon, cloud, or operating system. Among the projects under LF Edge, EdgeX Foundry stands out as an assembly of open-source microservices [96]. These microservices range from the Device Services Layer at the edge to the Core Services Layer at the center, facilitating deployment in microservices Docker containers. This setup enables users to run the system in containers and allows developers to

enhance or replace existing services. A key attribute of EdgeX Foundry is its interoperability, offering a platform that supports plug-and-play components. This not only enables developers to leverage existing connectivity standards but also contributes to unifying the marketplace. Overall, EdgeX Foundry provides a versatile software framework designed to fast-track the deployment of IoT solutions.

Eclipse Kura represents a project under the Eclipse IoT umbrella focused on developing IoT gateway solutions [98]. Its management is straightforward, primarily because it runs on the Java Virtual Machine (JVM), utilizing the OSGi bundle to streamline the construction of blocks, network configurations, and communications with IoT servers. Kura provides API access to the gateway's underlying hardware, including serial ports, USB, GPIOs, GPS, and more. By integrating with the Everyware Software Framework (ESF) – an enterprise-ready, multi-platform IoT Edge Framework designed for IoT Gateways – Kura enhances its capabilities, enabling the transmission of processed data to cloud platforms such as Everyware Cloud, Eclipse Kapua, among others.

Chapter 3

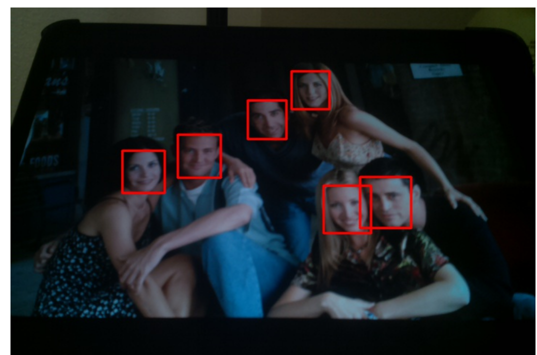
Pre-Work

This chapter outlines the preliminary work required to implement the Dynamic Distributed Scheduler, including device profile evaluation and AI applications in distributed systems.

3.1 Device Profile Evaluation

To meet our latency requirements, we need to determine the processing time for each task on devices with variable loads. Since containers are lightweight and are a suitable platform for deploying applications on edge networks, we conducted a test on the container to observe the impact of load on its performance.

(a) Original Picture



(b) Face Detection Result

Figure 3.1: An Example of Face Detection

3.1.1 Face Detection Container

Face detection is a computer vision task that involves locating and detecting human faces in images or videos. We use the Viola-Jones algorithm for the face detection container. The docker container image size is 1.08GB. When a picture with people is processed, the container detects all the faces in the image and highlights them with circles, as shown in Figure 3.1.

The runtime of the face detection algorithm can be influenced by several factors, including hardware. Faster processors and more memory can speed up the runtime of the algorithm. To demonstrate the runtime difference, we will use the same algorithm and image on two different devices, whose systems specifications are shown in Table 5.2 The size of the input image being processed is another factor that can influence the runtime of the face detection algorithm. The larger the input image, the longer it will take to process, as the algorithm needs to examine more pixels to detect faces, which requires more computation. Table 3.2 shows the runtime of the above-mentioned face detection container for different image sizes on the edge server.

Table 3.1: System Configuration

Component Name	**Specifications**
Edge Server	2.3GHz Dual-Core Intel Core i5, 8 GB RAM, 256GB Disk
Raspberry Pi	Quad core Cortex-A72 (ARM v8) 64-bit 8GB RAM, 1.8GHz Clock Speed
Smart Phone	Octa-core (4x2.3 GHz Mongoose, 4x1.6 GHz Cortex-A53), 4GB RAM

Table 3.2: Runtime for Different Image Size

Image Size (KB)	29	87	133	172	259
Runtime (ms)	223	417	615	798	1163

3.1.2 Profile Evaluation Scenarios

This section presents an experimental study to evaluate the processing time of containers in different scenarios under varying workloads. Based on the status of existing containers and the new container, there are four scenarios to represent the processing time of containers. In the first scenario, we started n containers and waited for them to warm up before sending a face detection request to all of them simultaneously. We recorded the start time (S) and the end time (E) of each container to calculate its processing time, where E-S is the time taken to process the request. We repeated this experiment 10 times for each situation.

In the second scenario, we recorded the start time (S) before starting to create n cold containers. We then calculated the processing time of each container by recording the start time (S_i), running the application, and recording the end time (E_i). We also calculated the total processing time by adding the time taken to create the containers to the processing time of the last container.

The third and fourth scenarios were designed to simulate real-world situations where an additional job needs to be allocated to n existing containers. In scenario three, we started n containers and an additional container simultaneously. When the n containers started processing the face detection request, we sent a request to the additional container. We recorded the processing time of each container and the additional container to analyze the impact of the new job on the existing ones. In scenario four, we repeated the same process as scenario three but created the additional container only when it received the request.

3.1.3 Profile Evaluation Results

Table 3.3: Profile of Cold Container on the Edge Server

# of containers	1	3	5	8	11
Current Container (ms)	63887	121766	226044	328269	716767
New Container (ms)	52554	71788	106596	165717	437846

Due to space constraints, this paper presents a subset of the experimental results. Tables 3.3

Table 3.4: Profile of Cold Container on the Raspberry Pi

# of containers	1	2	3	4	5	6
Current container (ms)	160802	198529	248812	313466	424130	520442
New container (ms)	168279	179280	188633	211136	241222	249413

and 3.4 display cold start container test outcomes for the edge server and Raspberry Pi, respectively. "Current Container" refers to the runtime in existing containers, while "New Container" details the runtime in a new container alongside existing ones, offering insights into cold start impacts on performance.

Table 3.5: Profile of Warm Container on the Edge Server

# of containers	1	2	3	4	5	6	7	8
Average Time (ms)	223	273	366	464	540	644	837	947
Total Time (ms)	11193	6930	6216	5951	5794	5507	6020	6099

Tables 3.5 and 3.6 display warm container test outcomes for the edge server and Raspberry Pi, respectively. Without startup latency, task processing times are significantly reduced. We evaluated the total processing time for 50 images, each 29KB, across varying numbers of containers, N. As the experiments revealed that the runtime in a new container, when added to N existing containers, closely matches the average runtime across $N + 1$ containers, we've consolidated these findings into a single table for clarity.

Based on the above results, we know that first of all, with more containers, average processing times increase due to the limited computing resources. Then for the total processing time for all tasks as shown in Table 3.5, increasing containers from 1 to 2 cuts the processing time for 50 images nearly in half, from 11193ms to 6930ms, utilizing dual-core CPU more efficiently. However, beyond 3 containers, CPU usage nears capacity, with idle CPU dropping close to 0%, leading to minimal time improvements. Consequently, performance plateaus around 6000ms as additional containers no longer enhance efficiency due to CPU load saturation.

The results affirm the approach of aligning the number of active containers with system workloads, establishing a strong basis for dynamic distributed scheduling strategies.

Table 3.6: Profile of Warm Container on the Raspberry Pi

# of containers	1	2	3	4	5	6
Average Time (ms)	597	613	651	860	1071	1290
Total Time (ms)	29934	15399	11072	11042	11043	11074

3.2 AI Applications Using Distributed Systems

To integrate with the dynamic distributed scheduler, we need to get the device profile for this augmented reality application. This device profile represents the computing capacity for a single application under varying dynamic CPU loads. This section will introduce how to evaluate it under different scenarios.

3.2.1 Methodology

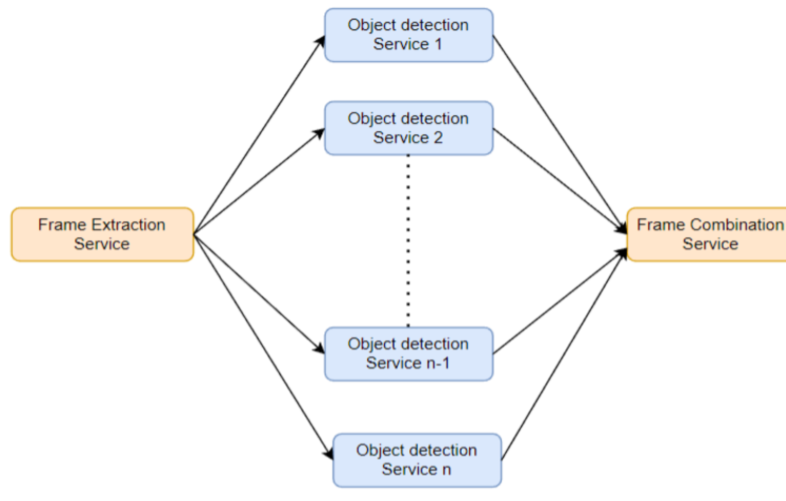


Figure 3.2: Distributed System Architecture

Our system operates with a division into three distinct services, each with its own set of responsibilities. The distributed system architecture is as shown in Figure 3.2

Firstly, we have the Frame Extraction Service, which is responsible for the conversion of video files into a series of images. These images are then sent to multiple object detection servers simulta-

neously via threading, depending on the number of servers available. Our mode of communication for sending the images to the object detection servers is REST where these images are pickled and sent as POST requests.

Next, we have the Object Detection Service, which houses the object detection models responsible for detecting objects in the series of images generated by the Frame Extraction Service.

We make use of SSD $MobileNet_v3$ to carry out the task of object detection with precision and accuracy. The SSD MobileNetV3[103] are neural networks designed for use on mobile devices such as smartphones. The new MobileNetV3 models have been developed through a blend of automated search algorithms and innovative architectural designs, tailored to excel in both high-resource and low-resource scenarios. The results show that MobileNetV3-Large is 3.2%, more accurate on ImageNet classification while reducing latency by 20% compared to MobileNetV2[**?**], and MobileNetV3-Small is 6.6% more accurate compared to a MobileNetV2 model with comparable latency. MobileNetV3-Large detection is over 25% faster at roughly the same accuracy as MobileNetV2 on COCO detection. MobileNetV3-Large LRASPP is 34% faster than MobileNetV2 R-ASPP at similar accuracy for Cityscapes segmentation. Overall, the new MobileNet models achieve state-of-the-art results for mobile classification, detection, and segmentation. Given our need for high speed and the fact that we will be utilizing Raspberry Pi, we have decided to employ the SSD MobileNetV3 model.

Finally, there is the Frame Combination Service, responsible for merging the images produced by various object detection services and presenting them to the user as a seamless live video stream. Pygame is utilized for implementation, where a fixed window size is maintained, ensuring images are correctly ordered and displayed to the user.

Each component within our system fulfills a vital role, ensuring the efficiency and precision of our object detection system. Through their collaborative efforts, these services facilitate seamless object detection, ensuring a superior user experience. Each service is encapsulated within a Docker container, and the entire system is interconnected through Kubernetes, enhancing its scalability and manageability.

3.2.2 Experiment Results

We have different environments where we performed our experiments and generated different results. In our experiments, we utilized various environments to produce distinct results. The specific configurations of the devices used are outlined in Table 3.7.

Table 3.7: System Configuration of Distributed AR Application

Component Name	Specifications
Mac	.6 GHz 6-Core Intel Core i7, 16GB RAM
sus TUF A15	AMD Ryzen 5 4600H with Radeon Graphics, 3000 Mhz, 6 Core(s)
Smart Phone	Quad core Cortex-A72 (ARM v8), D64-bit SoC @ 1.8GHz

3.2.2.1 Case 1: All on the edge server

Figure 3.3 shows how all services are connected and deployed on an edge server. By increasing the number of object detection containers to simulate dynamic CPU load, we get the average time for object detection.

Figure 3.4 shows the correlation between the number of object detection containers and the average time taken to detect objects in frames with different video quality. The data showcases a strong relationship between these two variables, and as the number of object detection containers increases, the time taken to detect objects in frames decreases, indicating optimal performance of the system.

When the video quality is 480p, the average time required for a single object detection container to detect objects is approximately 0.71 seconds. This time reduces significantly when we increase the number of object detection containers to two, with the processing time being reduced to 0.48 seconds. This improvement in processing time is quite remarkable, and it highlights the

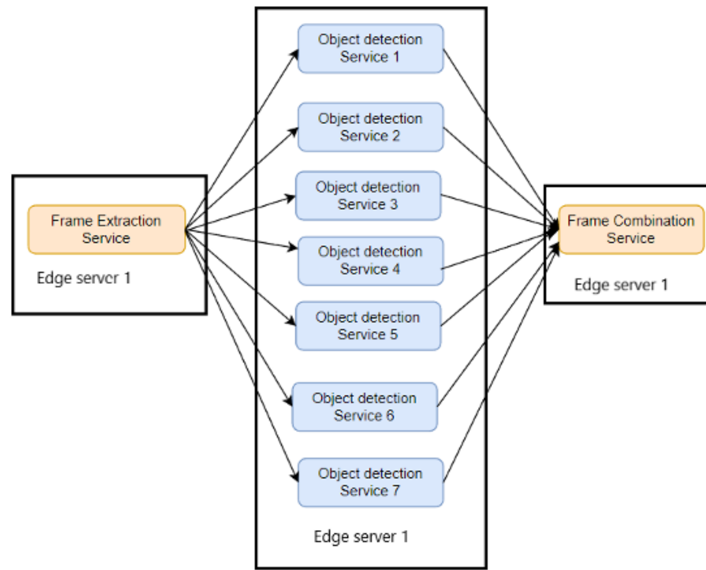


Figure 3.3: All On the Edge Server Architecture

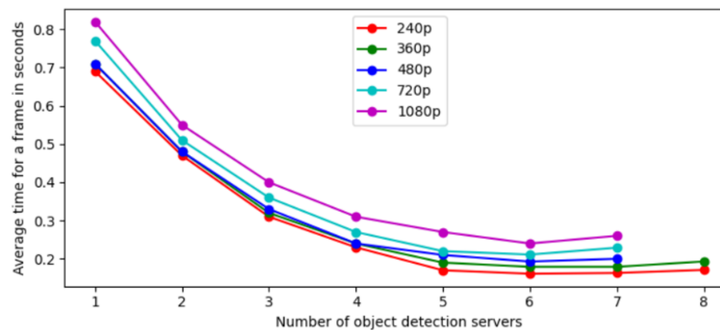


Figure 3.4: All on the Edge Server Results

advantages of utilizing multiple object detection containers to perform the task efficiently.

Moreover, as we continue to increase the number of object detection containers, the processing time required to detect objects decreases even further. For instance, with three object detection containers in use, the average time required to detect objects is reduced to 0.33 seconds. Similarly, when four object detection containers are used, the processing time required for object detection reduces even further to a mere 0.24 seconds.

This trend indicates that the system's processing capabilities can be effectively increased by adding more object detection containers, leading to a reduction in the time required to detect

objects in frames. However, this correlation between the number of object detection containers and the processing time is not infinite, and after reaching a certain threshold, adding additional object detection servers does not significantly decrease the detection time any further.

In fact, if we continue to add more object detection servers beyond that threshold point, the processing time increases, indicating that the system has reached its optimal efficiency level. Hence, it is crucial to identify the ideal number of object detection containers required for our system to operate efficiently.

As we delve deeper into the graphical data, we can focus our attention on the specific instance of 240p. In this scenario, we can observe that the optimal threshold is achieved when we have six containers. After this point, adding one more object detection container does not cause a significant decrease in processing time, and it is evident that the system has reached its maximum capacity.

Furthermore, we can also notice that adding additional object detection containers actually results in a slight increase in the time taken to detect objects, indicating that the system is beginning to reach its limit. Therefore, it is essential to carefully monitor and optimize the number of object detection containers used in our system to ensure efficient and effective performance.

3.2.2.2 Case 2: Distribute Across Computing Nodes

Figure 3.5 depicts the way to distribute object detection service across computing nodes. Considering that distributing service across different computing nodes adds extra communication costs, a careful balance between performance improvement and increased communication delays is needed to fully realize the performance benefits of task distribution.

3.2.3 Best Scenario

By increasing the number of containers on each computing node, we get Figure 3.6. The data illustrates a consistent trend across all video qualities—low, medium, and high—where the average processing time decreases with an increase in the number of object detection containers, ranging from one to twelve. This reduction implies that a greater number of containers enables

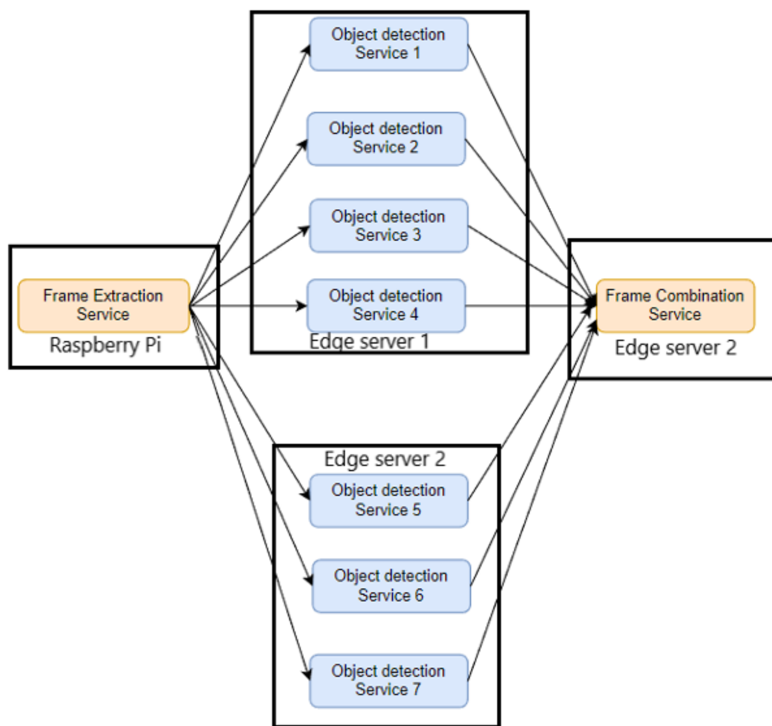


Figure 3.5: Distribute Across Computing Nodes Architecture

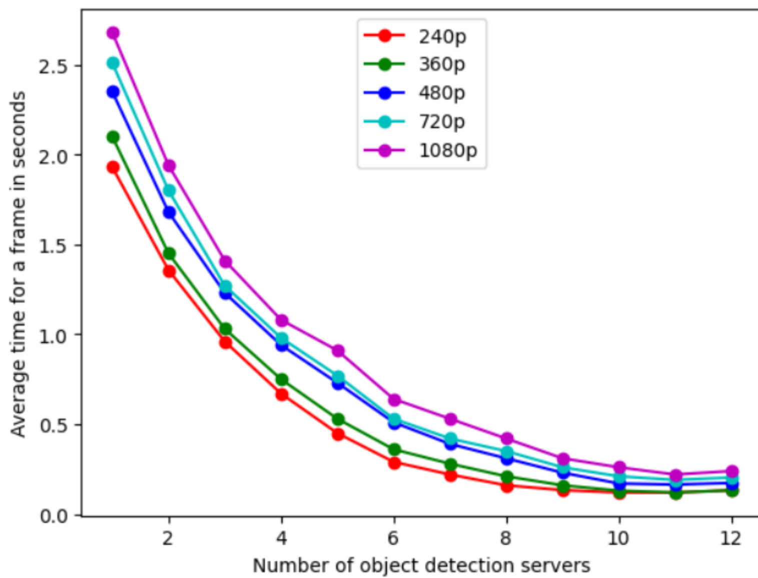


Figure 3.6: Distribute Across Computing Nodes Results

faster parallel processing of frames since each container handles a smaller segment of the video.

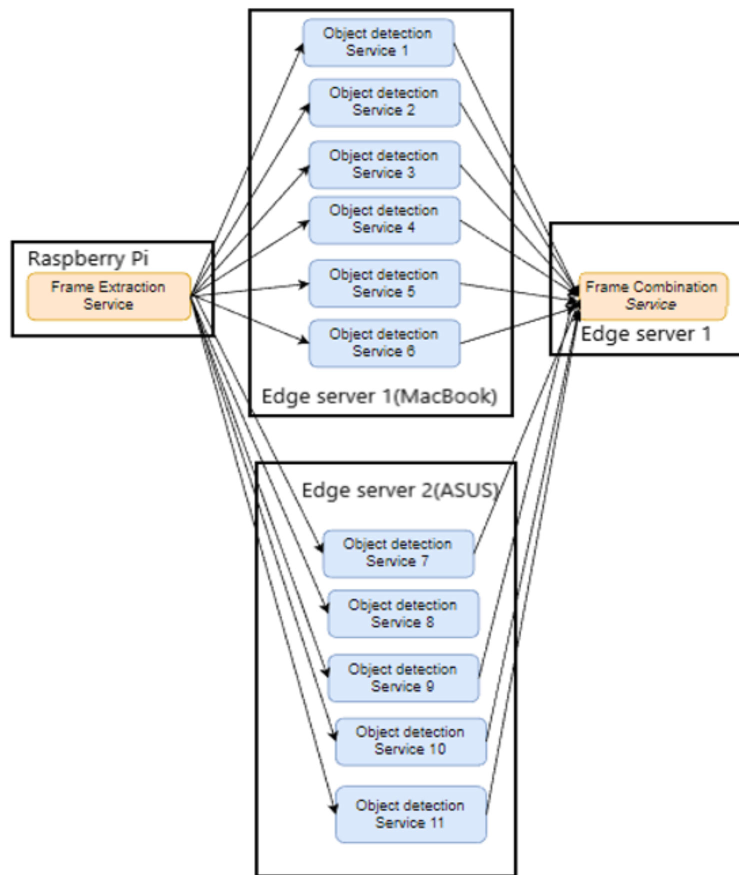


Figure 3.7: Best Output scenario

However, the graph also highlights a crucial observation: there exists a threshold for each video quality, beyond which adding more containers fails to further reduce the average processing time. Specifically, this threshold is met at ten containers for low-quality video and eleven containers for medium and high-quality video. Beyond to this point, exceeding these container counts leads to an increase in average processing time rather than a decrease. This phenomenon suggests that there is a limit to how much parallelization can enhance processing efficiency, indicating that other factors might come into play, potentially hindering the process's overall performance.

The quality of the video itself is a significant factor affecting our results. The graph clearly illustrates that as the video quality increases from low to medium to high, the average processing time substantially rises for each number of containers. This implies that higher-quality videos require more time for processing compared to lower-quality ones, regardless of the number of containers

available. There are several potential explanations for this observation. Firstly, higher-quality videos tend to have larger file sizes and resolutions, leading to increased transfer times between services via REST calls over the network. Secondly, these videos contain more intricate details and complex frames, prolonging the time required for frame extraction and combination. Additionally, higher-quality videos demand more computational and memory resources from the object detection model, leading to extended inference times and the production of accurate results.

When we have a Frame extraction service on Raspberry Pi and six object detection and one frame combination services on MacBook Pro and five object detection services on Asus, we get the lowest average time to detect frames and the time is 0.20.

Chapter 4

Dynamic Distributed Scheduler

4.1 Scheduler Architecture

This thesis addresses challenges in the edge environment by building a dynamic distributed scheduler that schedules different computing tasks of an application across different computing nodes based on the current computing and networking conditions to meet all the requirements of the application, including latency, privacy, power, and cost constraints. This scheduler constantly monitors the current compute and network conditions and dynamically adjusts the scheduling of the compute tasks accordingly. It has five important features:

(1) Device profiling: To address heterogeneity among different computing devices, each device in the IoT environment is profiled for different computing tasks under different computing loads to enable the scheduler to make optimal scheduling decisions at runtime.

(2) Predictive profiling: To address the limitation of pre-profiling every possible device under every possible computing environment, the scheduler incorporates a prediction mechanism to predict the running times of various computing tasks during runtime and adjusts schedules accordingly.

(3) Two-level scheduling decision: To minimize the overhead of the scheduler, the scheduling decision is done at two levels, a primitive scheduling decision at the low-end devices and a near-optimal scheduling decision at the edge server.

(4) Local compute principle: To address the uncertainty in determining accurate communication latency under every possible computing environment, the scheduler schedules tasks locally as a low-level scheduling decision as long as this scheduling satisfies all the application requirements.

(5) Predictive compute drops: The scheduler selectively drops some compute task instances based on application guidance to maximize meeting the overall application constraints.

4.1.1 Scheduler Design Principles

As mentioned earlier, an IoT environment at the edge is typically highly hybrid and dynamic in nature. Computing devices range from low-end, resource-constrained devices to moderately powerful edge servers. Further, the communication and compute load can vary significantly over time depending on what events take place at the edge. So, it is important to ensure that the scheduler takes into account these dynamically varying factors while deciding task scheduling. Consequently, a key challenge in building a distributed scheduler for a dynamic edge environment is how to balance the overhead incurred by maintaining current up-to-date information about the computing environment and scheduling decision-making with the potential gains in application performance.

Our scheduler distributed architecture consists of *two levels*. The goal of the lower level is to maintain up-to-date information about the computing environment in terms of current communication latencies between different end devices and the edge server, the current CPU load of all devices at the edge including the edge server, the remaining battery power of the devices, etc. This level monitors the computing environment at regular intervals and keeps updating the up-to-date information. The complete up-to-date information is stored at the edge server, while only local compute information is stored at the end device, i.e. a device stores only its CPU load, remaining battery power, and the communication latency between itself and the edge server, while the edge server store CPU loads and remaining battery power of all devices, as well as communication latencies between every end device and itself. This level is implemented by a set of processes, one each

running on each end device and the edge server. At regular intervals, a process running on an end device sends the current CPU load and remaining battery power of the device to the edge server. In addition, it measures round-trip latency (using ping messages) between the edge server and itself and sends that latency to the edge server.

The goal of the higher level is to make appropriate scheduling decisions based on the computing information maintained by the lower level. Each node, including edge servers and end devices, runs a scheduler component. The scheduler component running at an end device makes a scheduling decision purely based on the current local compute information provided by the lower level and the *device profiling* information (discussed in the next subsection) of the device. The scheduler component running at the edge server makes a scheduling decision based on the complete compute information provided by the lower level and device profiling information of all devices.

When a new compute task arrives at an end device, the goal of the scheduler component running on that end device is to simply decide whether to run the task locally or send it to the edge server. Our scheduler uses a *local compute principle* to make this decision. If all the constraints of the computing task can be satisfied by running the task locally based on the current local compute information and the device profile information, the scheduler schedules the task to run locally. Otherwise, it sends that task to the edge server. On the other hand, the scheduler component at the edge server runs a multi-objective optimization algorithm based on current (global) compute information and device profiles of all devices. For tasks allocated to the edge server for processing, the scheduler may discard tasks that cannot be processed while meeting all constraints, thereby maximizing overall performance.

4.1.2 Device Profiling

Every compute task can have a set of constraints, such as a real-time constraint for task completion, privacy constraint with respect to whether the data can be shared with other devices, cost constraints with respect to some max dollar amount, etc. In order to decide whether all these constraints can be satisfied while running a task on a particular device, we need to know a priori

the time it would take and/or the power it would consume, to run the task on the device under the current compute conditions (CPU load, communication latency, etc.). In the first version of our scheduler, we profile in advance each device for each type of task it would run under several different compute conditions. We call this the device profile information. This information along with the current compute condition is then used by the scheduler to make appropriate scheduling decisions. In this subsection, we describe how we profile the running times of a compute task under different compute conditions for different devices. Equation 4.1 indicates the total processing time for each task.

$$T_{total} = T_{trans} + T_{queue} + T_{process} + T_{reply} \tag{4.1}$$

Here, T_{trans} and T_{reply} represent the total data transmission and reply times respectively. T_{queue} denotes the queuing time of the task on the device. These metrics are obtained using multi-threading programming. However, determining $T_{process}$, the processing time, proves challenging. The complexity arises due to the different types of computing involved in different tasks, the heterogeneity of different devices, and the variable workloads they experience.

Since a microservice-based architecture using containers has become the standard of structuring IoT applications at the edge, we operate under the premise that all AI applications are deployed within containers. Docker is designed for a single application per container approach, ensuring containers are loosely coupled with one another.

Current research [72] measures container performance benchmark tools that simulate adjustable loads on the system, including CPU, memory, I/O, and disk operations. However, it is hard to metric $T_{process}$ under variable workloads. To address this challenge, we suggest utilizing device profiling to pre-evaluate a comprehensive profile table that includes the number of active containers as a measure of workload. During scheduling, the scheduler can then use the container count to retrieve the corresponding $T_{process}$ for informed decision-making. Refer to Chapter 3 for more details about device profiling.

4.1.3 Predictive Profiling

A key challenge in device profiling is acquiring comprehensive profiling information in advance under every possible workload, which is highly time-consuming and sometimes even infeasible. The goal of predictive profiling is to estimate the running times of various compute tasks on a device under different workloads (number of containers running concurrently) based on the amount of time it takes to run a single task container with other containers running concurrently. Equation 4.2 shows how we do this estimation. In this equation, notation T_{ym} indicates the amount of time it would take to run a new container for task y while there are m containers of task y already running at present. The equation illustrates an estimate of the amount of time it would take to run a new task x container while there are m task x containers already running concurrently. This estimate is based on first computing a multiplying factor $(T_{ym} - T_{y1})/T_{y1}$ for a candidate task y using actual experiments.

$$T_{xm} = T_{x1} * (T_{ym} - T_{y1})/T_{y1} \tag{4.2}$$

4.1.4 Architecture Components

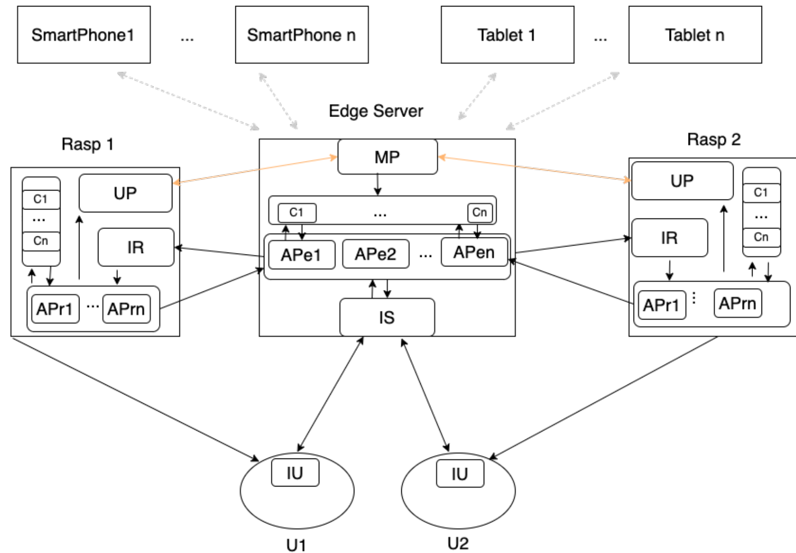


Figure 4.1: Architecture

The proposed architecture, illustrated in Figure 4.1, is tailored to support multiple Edge AI applications (APe1 ... APen) by distributing tasks to end devices based on their proximity, ensuring efficient data acquisition. The users that need to avail of the services available in an IoT environment first connect with the environment by sending an appropriate request to the edge server. Upon receiving an application request from a user, the edge server identifies nearby end devices and delegates tasks accordingly. For example, in a crowded mall scenario, the edge server activates end devices in close proximity to the user, such as cameras or video modules, to capture a stream of the crowd and enhance target person detection accuracy. This architecture is comprised of three major components, namely the edge server, end devices, and mobile users. The edge server, which acts as the central coordinator, consists of four key components:

- Interface Server (IS): The IS receives user requests, checks whether the user is authorized to access the application (say x) being requested, and then forwards the request to the specific application manager (APex) of application x.
- Applications on Edge Server (APe): An edge server may support multiple edge AI applications, APe1 to APen, these correspond to different AI application managers and their scheduling mechanisms.
- Maintain Profile (MP) Module: This module interfaces with other Update Profile (UP) modules to gather and maintain a global profile table with information from all end devices via the lower-level component of the scheduler. Each APe is aware of this update through shared memory.
- Container Pool: Each APe is associated with a specific container pool, ranging from C1 to Cn. Figure 4.1 illustrates containers linked to one APe.

Any device that has computing resources, such as Raspberry Pi's, SmartPhones and Tablets, can serve as an end device within this architecture. Before joining the system, the device needs to

be certified to ensure it meets the requirements and is profiled. Each end device is comprised of an Interface Receiver (IR), a container pool, and the following two components.

- Update Profile (UP) Module: This module communicates the device's profile information, including the device ID, CPU load, remaining battery power, communication latency, etc to the edge server at regular intervals.
- Applications on Raspberry Pi (APr) Pools: Similar to APe, the APr pools, ranging from APr1 to APrn, correspond to various AI application managers and their scheduling mechanisms.

Mobile users submit requests that include details such as specific applications, locations, and time constraints. Upon establishing a connection with the system, they await a response, facilitated by multi-threading.

4.1.5 Architecture Workflow

Figure 4.1 depicts the architecture of a distributed scheduler system. The system receives a request from a user through IS that includes the requested application ID, user location, and other relevant information. IS forwards the request to the corresponding APe, say APex such as a face detection application.

Based on the user's location, the APex determines which end device to get the source data from. The chosen end device is the one connected to the appropriate sensor needed by the application, e.g. a Rasberry Pi connected to a camera. It sends a message to that end device to start the requested application and retrieve the appropriate sensor. APex then sends a reply to the user. Assuming Rasp 1 in Figure 4.1 is the chosen end device, the workflow and corresponding pseudo-code are presented in Algorithm 2. To improve efficiency, multi-threading is used in this program.

The IR on Rasp 1 receives the request from the edge server and sends it to the APr. For each task, the APr makes scheduling decisions based on the current load. It gets load information from the UP module, which collects and updates a table reflecting the current load. This module also

Algorithm 1 The Dynamic Distributed Scheduler Algorithm

Input: profile tables of all computing nodes, time constraint T, original video
Output: number of frames meeting time constraint, video with augmented reality effect

```
1:  t_pr1                                  ▷ predictive running time on R1
2:  t_pr2                                  ▷ predictive running time on R2
3:  t_pes                                  ▷ predictive running time on the edge server
4:  t_cr1                                  ▷ Current time constraint on R1
5:  t_cr2                                  ▷ Current time constraint on R2
6:  t_ces                                  ▷ Current time constraint on the edge server
7:  t_q1                                   ▷ Queuing time on R1
8:  t_qes                                  ▷ Queuing time on the edge server
9:  t_r1es                                 ▷ Communication delay from R1 to the edge server
10: t_esr2                                 ▷ Communication delay from the edge server to R2
11: R1_PRO(num)                            ▷ Function for predictive running time on R1
12: R2_PRO(num)                            ▷ Function for predictive running time on R2
13: ES_PRO(num)                            ▷ Function for predictive time on the edge server
14: t_pr1 ← R1_PRO(num)
15: t_cr1 ← T − t_q1
16: if t_pr1 < t_cr1 then
17:     Process this frame locally
18: else
19:     Send this frame to the edge server
20:     t_pr2 ← R2_PRO(num)
21:     t_cr2 ← t_cr1 − t_r1es − t_esr2
22:     if t_pr2 < t_cr2 then
23:         Send this frame to R2
24:     else
25:         t_pes ← ES_PRO(num)
26:         t_ces ← t_cr1 − t_r1es − t_qes
27:         if t_pes < t_ces then
28:             The edge server processes this frame
29:         else
30:             Drop this frame
31:         end if
32:     end if
33: end if
```

monitors the network condition and updates it. If the end device can process it locally satisfying all constraints, the APr sends it to the local container. If not, the image is sent to the APex on the edge server. This is the lower-level scheduling on the end device, which ensures the maximum usage of the source end device's capacity.

The APex on the edge server receives tasks from Rasp 1 and makes globally optimal decisions based on the global profile. For example, if R2 in Figure 4.1 is capable of processing the task, it is sent to R2; otherwise, the function ES_PRO(num) is called to decide whether to drop or continue with the task. This is the upper-level scheduling with the coordinator scheduler, which optimizes the overall performance and maximizes usage of other end devices' capacity. We note that there have been a large number of multi-objective optimization algorithms proposed for distributed scheduling and APex can use any of those algorithms to make appropriate scheduling decisions. The key advantage of our scheduler is that it provides accurate values of the parameters such as CPU load, running times, etc for the current computing conditions that these algorithms use. So, the scheduling mechanism presented here is complementary to these approaches. The primary objective of this paper is to validate the efficiency of the dynamic distributed scheduler based on device profiling and dynamic monitoring.

4.2 Experiment with an AI Application

4.2.1 An AI Application

The experimental application is illustrated in Figure 4.2. It provides a visual representation of the components of the architecture and their interactions. The architecture includes an edge server, two Raspberry Pis, a mobile phone, and a camera. Each picture taken by the camera, would be sent to R1, and the scheduler system makes decisions of where to process it.

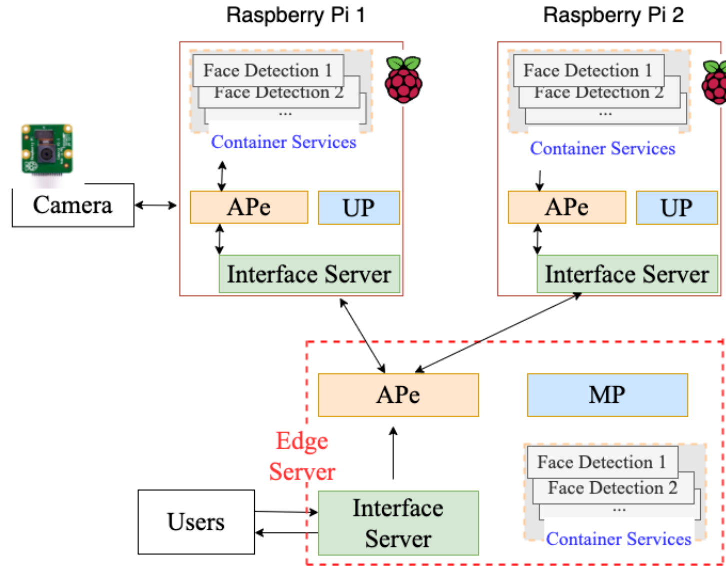


Figure 4.2: Experiment Application

4.2.2 Results and Analysis

In order to compare our Dynamic Distributed Scheduler (DDS) with other approaches, we created three comparison groups. The first group is to run ALL On the Raspberry Pi (AOR), without utilizing any edge server computing resources. The second comparison group involved transmitting all images to the edge server and Running All on the Edge Server (AOE). We created a third comparison group where the Raspberry Pi was responsible for processing images with odd-numbered sequences, while all images with even-numbered sequences were transmitted to the edge server for processing, named Even Odd Distributed Scheduling (EODS).

4.2.2.1 Set 50 images

This experiment explores the performance of four different scheduling algorithms when an end device continuously receives a stream of images from a buffer module after receiving an application request. The evaluation results of testing 50 images with each scheduling algorithm are presented in Figure 4.3.

As the time interval between two images can significantly impact the performance of the

system, the experiment sets it as a parameter and tests the algorithms' performance at four different time intervals: 50ms, 100ms, 200ms, and 500ms. The evaluation results for each time interval are presented in Figure 4.3a, Figure 4.3b, Figure 4.3c, and Figure 4.3d, respectively. Each subfigure in the evaluation results reflects the relationship between the number of images that meet the requirements and time constraints.

As shown in Figure 4.3, we can observe that with an increase in time constraints, a greater number of images meet the time requirements for all scheduling algorithms. For instance, in Figure 4.3a, when the time constraint is set to 500ms, none of the images meet the constraint while running all images on Raspberry Pi. However, when the time constraint is increased to 5000ms, 30 out of the 50 images meet the requirement. Furthermore, we can observe that as the time interval increases, more images can meet the requirements. For example, when comparing the results of running all images on Raspberry Pi with a time constraint of 1000ms in Figure 4.3a and Figure 4.3d, only 3 images meet the requirements when the time interval is set to 50ms, but when the time interval increases to 500ms, all images can meet the requirements. This trend is reasonable because a longer time interval leads to a shorter queuing time and fewer simultaneous running containers, which reduces the computing load and decreases the processing time for each application as images are processed more efficiently.

From Figure 4.3, insightful conclusions can be drawn regarding scheduling algorithm design and building scheduling systems.

- It is important to set the minimum time constraint required for all requests. If the time constraint is too short, none of the scheduling algorithms can improve performance. For instance, when the time constraint is less than 200ms, none of the four scheduling algorithms meet the image processing requirements. Therefore, any application requests with a time constraint less than this time should be rejected.

- Computing nodes with powerful resources outperform weaker ones, as observed in all subfigures of Figure 4.3, where the edge server always performs better than the end device.

This trend is also observed in the device profile evaluation section.

- For requests with loose constraints such as long time intervals and long time constraints, running everything on the edge server has the best output and the least overhead.

- Distributed scheduling systems such as Even Odd distributed scheduling and the Dynamic Distributed Scheduler perform better than running all images on either the edge server or the end device. This observation is supported by all the cases in the experiment.

- The Dynamic Distributed Scheduler is better than the Even Odd Distributed Scheduler, except when the edge server is heavily loaded. For example, in Figure 4.3a, when the time constraint is 500ms, the Raspberry Pi is unable to process any image within the time limit, leading to a heavy load on the edge server and a long queuing list. In this case, the Dynamic Distributed Scheduler is similar to running everything on the edge server.

4.2.2.2 Set 1000 images

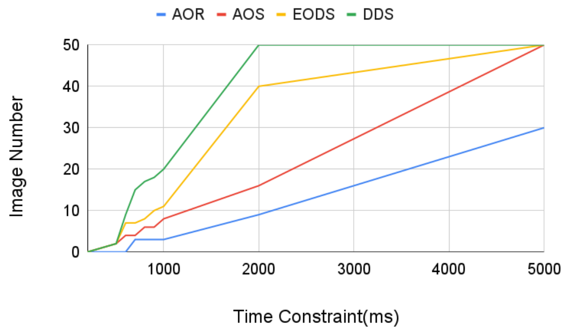
We further explore DDS's performance with continued streaming frames with 1000 images as shown in Figure 4.4. The evaluation results for a time interval of 50ms and 100ms are shown in Figure 4.4a and Figure 4.4b, respectively.

A comparison of the DDS algorithm with the other algorithms indicates the following observations:

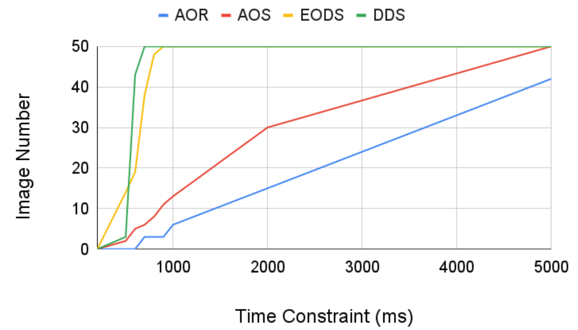
- DDS performs better when the time constraint is less than 30000ms for a time interval of 50ms and when the time constraint is less than 10000ms for a time interval of 1000ms.

- However, when the time constraint is not stringent, such as when the time constraint is greater than 60000ms for a time interval of 50ms and when the time constraint is greater than 30000ms for a time interval of 100ms, DDS does not perform as well as EODS.

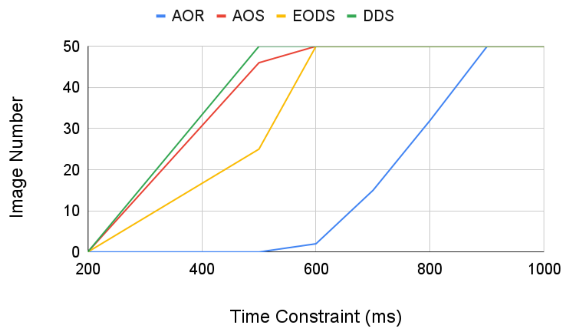
- The overhead in the DDS algorithm exists because the scheduler on Raspberry Pi saves



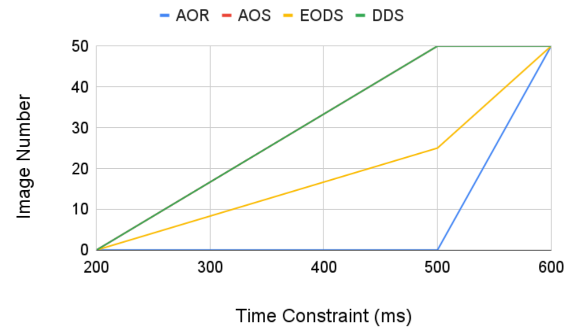
(a) Time Interval (50ms)



(b) Time Interval (100ms)



(c) Time Interval (200ms)



(d) Time Interval (500ms)

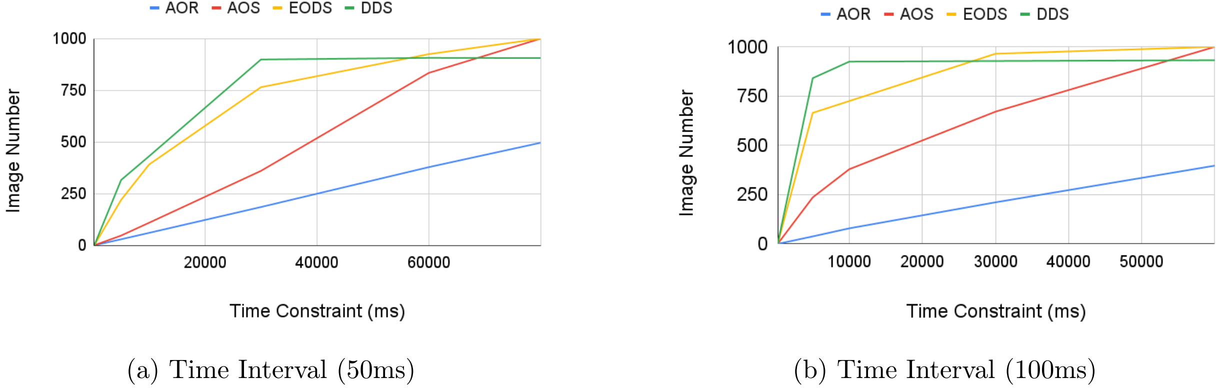
Figure 4.3: 50 Images

more images to process locally with more time, leading to longer queuing lists and many last images exceeding the time limit.

- In practical situations where the time interval and the time constraint are not large, DDS has the highest priority among all the scheduling algorithms for Edge AI systems.

4.2.2.3 Performance of extending the end device scale

In certain cases, the edge server may face difficulties in processing all images received by it. The graph depicted in Figure 4.3 highlights that when there are too many images waiting to be processed on the edge server within a limited time, the processing time of the last few images tends to exceed the given time constraint. To tackle this issue, one approach is to expand the scale of end



(a) Time Interval (50ms)

(b) Time Interval (100ms)

Figure 4.4: 1000 Images

devices, allowing the main scheduler on the edge server to dynamically distribute tasks across other end devices, thereby enabling more images to meet the time constraint. Another reason to offload tasks from the edge server is to maximize the utilization of end devices' computing resources while keeping the edge server lightly loaded to effectively manage the entire system.

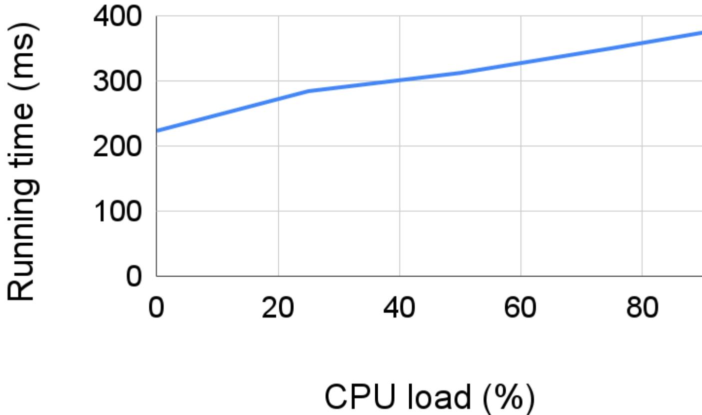


Figure 4.5: Relationship between CPU Load and performance

We set different CPU loads for the edge server to simulate its real-world scenarios where it is always occupied by many tasks. And the relationship between CPU load and performance is as show in 4.5. Firstly, we set an experiment to investigate the relationship between CPU load and the average processing time of containers. The experiment results showed that as the CPU load

increases from 0 to 100%, the average processing time of the containers also increases, suggesting that CPU load directly impacts container performance. At a CPU load of 0%, the container took an average of 223 units of processing time, which increased to 284, 312, and 350 units of processing time at 25%, 50%, and 75% CPU load, respectively. When the CPU load was close to 100%, the processing time increased to 374 units, indicating that the container was under significant strain at high CPU loads.

Merely increasing the number of end devices without a well-designed scheduling mechanism can lead to suboptimal performance and wasted resources. Therefore, it is crucial to prioritize the development of effective scheduling mechanisms in order to fully realize the potential benefits of edge computing.

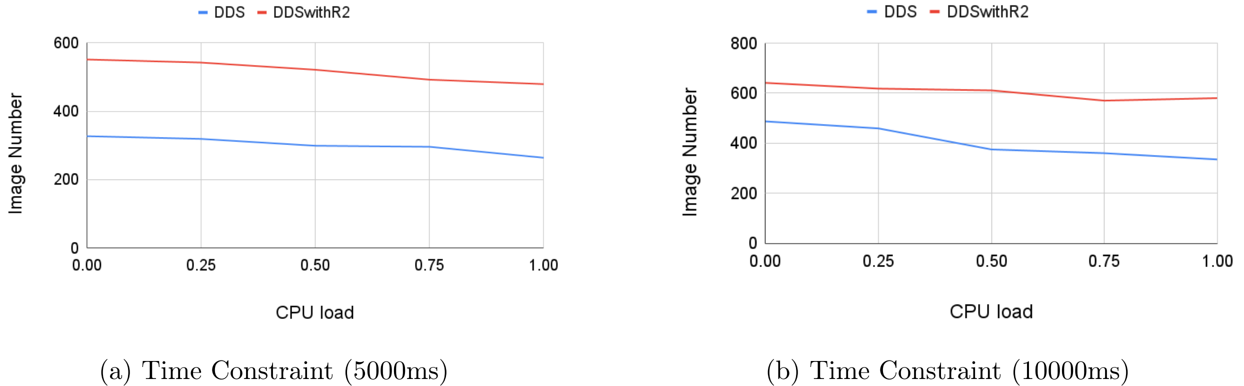
This paper proposed a way to extend DDS and allow end devices to update their profile, such as the number of containers running with the edge server, on a regular basis. By analyzing the number of containers running and running time under specific loads from container evaluation results, the edge server calculates the predicted processing time for the current image, enabling a queuing list of tasks on the end device. This enables the edge server to make dynamic decisions about current tasks that meet requirements and optimize performance by processing as many tasks that meet requirements as possible simultaneously.

However, using a queue to store tasks introduces a time difference between decision-making and actual execution, which can reduce predicting accuracy. To mitigate this issue, the scheduler checks whether the end device has available containers and only offloads the task to that device if containers are available.

Figure 4.6 presents a comparison between an edge server using DDS and an extended system that includes one more Raspberry Pi using the same DDS. The time interval is set to 50ms. Figure 4.6a and Figure 4.6b depict the performance under time constraints of 5000ms and 10000ms, respectively. To evaluate the system, we stress the CPU load from 0 to 100 percent and the number of images meeting the time constraint indicates its performance.

These results provide valuable insights into the performance of the DDS and its extension, as

well as their ability to handle varying levels of CPU load while maintaining task completion within specified time constraints.



(a) Time Constraint (5000ms)

(b) Time Constraint (10000ms)

Figure 4.6: DDS vs DDSwithR2

- The number of images meeting the time constraint decreases as the CPU load increases.
- The performance of the system improves when an additional Raspberry Pi is incorporated. For instance, in Figure 4.6a, at a CPU load of 0, the number of images meeting the time constraint is 327 when using DDS alone, whereas this number increases to 551 when using DDS with an additional Raspberry Pi, indicating an improvement of 69%.

4.3 Experiment with an AR Application

4.3.1 An Augmented Reality Application

Augmented Reality (AR) merges computer-generated content with live video, enabling the addition of virtual elements to real-world scenes through a device's camera. This technology is applied in various domains, such as security, video conferencing, and healthcare, with face detection in streaming videos being a notable use case [104].

The application operates in three phases: frame extraction, face detection, and frame reassembly into the original sequence. Aimed at real-time processing within edge networks, tasks are allocated based on resource intensity, with frame extraction at the end device connected to

the camera and face detection distributed across nodes for efficiency. This setup requires a robust scheduling system to ensure seamless application performance by effectively managing computing resources across the network, particularly focusing on the scheduling of face detection tasks within the application. Device profiling across various face detection warm container counts and video resolutions is done, as we introduced in Chapter 3. Due to space constraints, these details are omitted from this paper.

4.3.2 Set Up

Table 4.1: Devices Profile with 1080p Video

# of containers	1	2	3	4
Edge Server (ms)	681	1171	1837	2527
Rasp (ms)	3055	5898	8744	×

Incorporating the AR application within the architecture depicted in Figure 4.1 starts with the user sending a request to the edge server. The edge server forwards the request to R1, which then captures video through its camera for processing. R1 manages a frame extraction container and a frame combination container while running three face detection containers. Frames are either processed locally or sent to the edge server, where additional face detection is performed by both the edge server and R2. Finally, processed frames are recombined, and the application's response is relayed back to the user. All device profile evaluations have been completed. Due to space limitations, we have included only the results for the 1080p video in Table 4.1. The edge server has four warm containers awaiting requests, while the Raspberry Pi has three warm containers ready for requests.

4.3.3 Results and Analysis

This section details experimental outcomes across different video resolutions, illustrated in Figures 4.7, 4.8, and 4.9 for 240p, 480p, and 1080p videos, respectively. To assess the performance of our scheduler, we compare its performance with four static scheduling mechanisms: 1) All_R1:

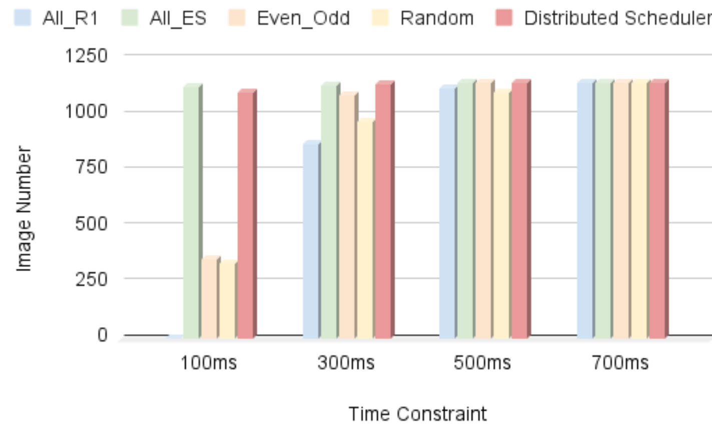


Figure 4.7: Experiment Results with 240p Video

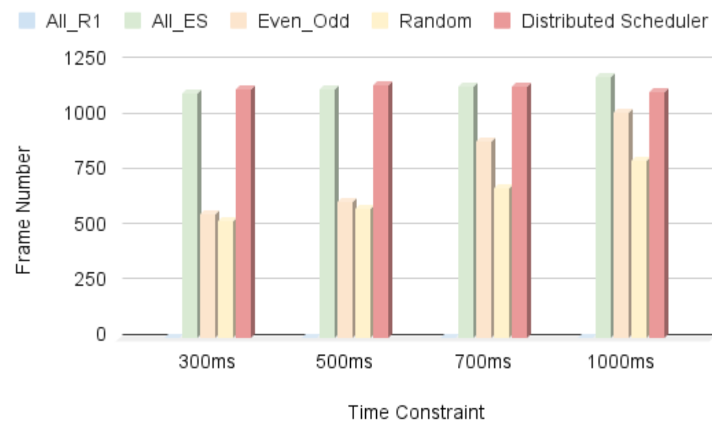


Figure 4.8: Experiment Results with 480p Video

Object detection of all frames is done locally on Rasp 1; 2) (All_ES): Object detection of all frames is done on the edge server and no object detection containers run locally on Rasp 1; 3) Even_Odd: Object detection of even-numbered frames are done locally on Rasp 1 and odd-numbered frames on the edge server; and 4) Random: Object detection of frames are randomly distributed between Rasp 1 and the edge server. Figure 4.7 shows the number of frames successfully processed (extraction, object detection, and combination) for a 240p video within four different real-time constraints of 100ms, 300ms, 500ms, and 700ms. This video contains a total of 1145 frames. We make the following observations:

- As expected, for higher values of time constraints, all algorithms successfully process a

higher number of frames.

- Our dynamic distributed scheduler outperforms single-computing-node algorithms, All_R1 and All_ES, with notable efficiency against All_R1. As Table 4.1 shows, the edge server's superior processing power enhances success rates for frame processing. Although All_ES approaches the dynamic distributed scheduler's performance, the edge server's role in managing AI applications and device profiles underscores the need to reduce its workload. Thus, the dynamic distributed scheduler offers a more viable and efficient strategy than All_ES.

- Our dynamic distributed scheduler outperforms Even_Odd and Random algorithms by optimizing end-device capacities to process more frames within time limits. While all algorithms perform well at generous time limits, like 700ms, tighter constraints reveal their limitations. For example, at a 100ms limit, Even_Odd and Random algorithms process only 355 and 336 frames, respectively, whereas the dynamic distributed scheduler successfully processes 1102 out of 1145 frames, showcasing its superior efficiency under realistic conditions.

Figures 4.8 and 4.9 present the experimental results for videos consisting of 1145 frames at 480p and 1080p resolutions, respectively.These results not only support but also further substantiate the observations mentioned earlier. Specifically, the dynamic distributed scheduler outperforms the other four algorithms across a range of time constraints, from tight to generous. As the resolution of frames increases, the difference in processing time for each frame between the Raspberry Pi and the edge server becomes more pronounced.

Figure 4.10 shows the end-to-end processing times to gauge the effect of various scheduling approaches for a 480p video with 1145 frames. The results show All_R1 algorithm takes 697 seconds for total processing, while other algorithms average 495 seconds, marking a 41% overhead increase with exclusive Raspberry Pi use. This implies the system incurs less overhead transferring frames to the edge server than processing solely on the Raspberry Pi.

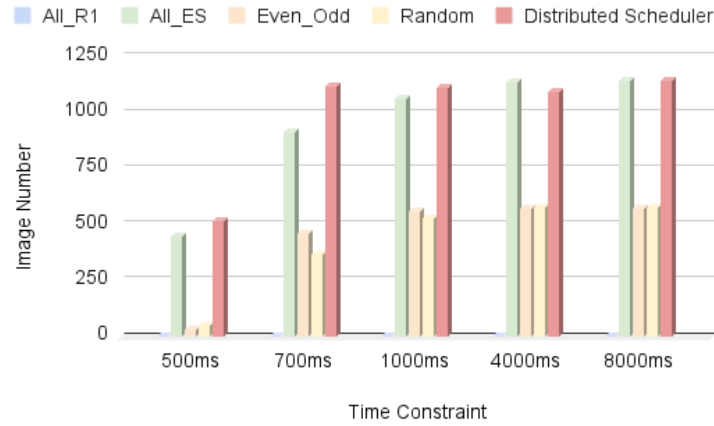


Figure 4.9: Experiment Results with 1080p Video

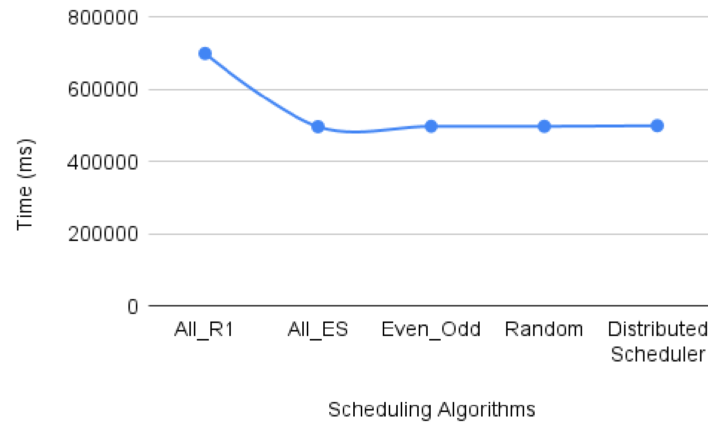


Figure 4.10: End-to-End Time

4.3.4 Analysis

The results highlight the dynamic distributed scheduler's ability to dynamically allocate computing tasks within the edge network, leveraging device profiling to enhance edge node utility and reduce edge server workload. In the following section, we delve deeper into showcasing the efficiency of the dynamic distributed scheduler by incorporating R2. Additionally, we introduce two optimized strategies aimed at addressing the scalability and adaptability challenges.

4.4 Optimization

We optimize the dynamic distributed scheduler for the following two aspects: first, we propose a predictive profiling solution to avoid completed profile evaluation. Second, we extend the dynamic distributed scheduler to accommodate more computing nodes and further optimize the algorithm.

4.4.1 Predictive Profiling and Dynamic Update

With guidance in Section III Subsection B predictive profiling, leveraging 240p profiling data, we predict 1080p face detection times under various conditions. For example, adding a container to a 240p video raises its processing time from 270ms to 510ms (an 88% increase). Similarly, a 1080p video's time is expected to jump by 88% from 3055ms to 6232ms with an additional container. Table 4.2 summarizes these predictions.

Table 4.2: Predictive Profile

# of containers	0	1	2
Predictive Time (ms)	3055	6032	8112
Actual Evaluation Time (ms)	3055	5898	8744

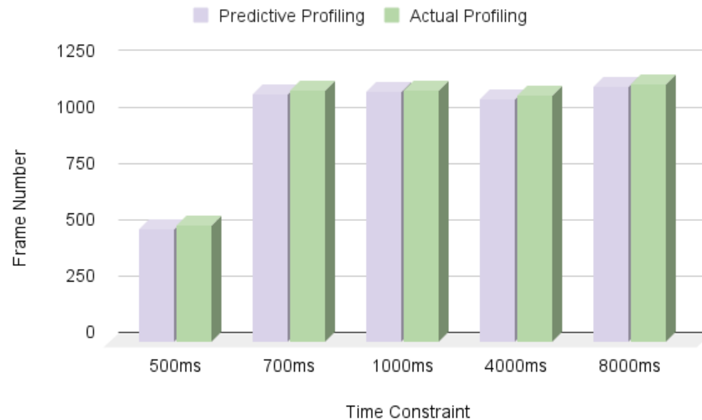


Figure 4.11: Predictive profiling vs Actual Profiling

Table 4.2 shows that the predicted times align well with actual times. To improve precision, we do runtime profiling as well. In this approach, we start with the predicted profile and then monitor

the execution times during runtime to update the profile dynamically. For example, initiating a new container updates the profiling for M active containers with recent data. Figure 4.11 compares actual and predictive profiling for a 1080p video of 1145 frames under various time constraints, revealing similar success rates between both methods.

4.4.2 Balance Between the Load on the Edge Server and Success Rate

4.4.2.1 Additional End Devices

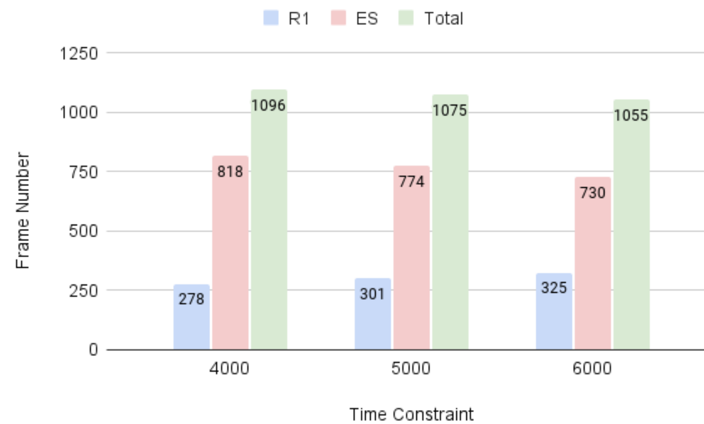


Figure 4.12: Dynamic Scheduler Algorithm

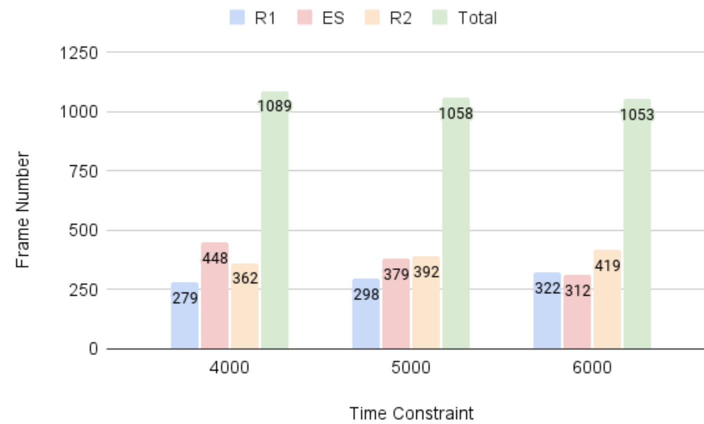


Figure 4.13: Extend Dynamic Scheduler Algorithm

We now evaluate our scheduler for a case where there is more than one end device (two Rasberry Pi's) and the edge server. Figure 4.12 shows a 1080p video's processing by a system with one

Raspberry Pi and edge server, under 4000ms to 6000ms constraints, highlighting each component's performance. Figure 4.13 compares this with the dynamic distributed scheduler algorithm enhanced by Raspberry Pi 2 (R2). Comparing those two figures, we observe:

- With higher time constraints, more frames are directed to end devices and fewer to the edge server. Without R2, R1 processes 327, 371, and 415 frames under 4000ms, 5000ms, and 6000ms constraints, respectively. With R2, these numbers slightly change on R1 to 328, 368, and 412, while R2 processes an additional 371, 398, and 421 frames for each respective time constraint. This shift indicates that, as time constraints relax, both R1 and R2 are increasingly utilized for frame processing, optimizing the distribution based on the scheduler's assessment of local processing capabilities and the edge server's evaluation of R2's capacity.

- While R2 reduces the edge server's workload, its effect on overall performance is slight, with marginal differences in frame processing—1096 vs. 1089, 1075 vs. 1058, and 1055 vs. 1053—for time constraints of 4000ms, 5000ms, and 6000ms, respectively.

- Increasing the time constraint from tight to medium improves the frame success rate, but a further increase of time constraints results in decreased success rates. In the basic setup, R1's success rate drops from 85.0% (278 out of 327 frames) at 4000ms to 78.3% (325 out of 415 frames) at 6000ms. The system's overall rate similarly declines from 95.7% (1096 out of 1145 frames) to 92.1% (1055 out of 1145 frames). The extended setup showed that allocating more frames to R1 or R2 lengthens the queue, raising the chance of time-constraint failures. The system's multi-threaded design and the challenge of precise run order management often lead to inevitable queue growth. Additionally, success rates drop due to scheduling timing; initial local allocations are based on the current number of active containers, but as the frame queue expands and more containers become available, additional containers start running. This discrepancy in timing can negatively impact the final performance.

4.4.2.2 CPU Load Variation on the Edge Server

Table 4.3: Adjusted Device Profile

# of containers	0	1	2
Actual Evaluation Time (ms)	3055	5898	8744
10% Increase (ms)	3626	6264	8744
30% Increase (ms)	4761	6815	8744
50% Increase (ms)	5900	7367	8744

The observations indicate that as time constraints are increased, more frames are directed to the end devices. However, this results in longer frame queues on these devices, leading to a decrease in both the success rate of frames processed by the end devices and the overall number of frames meeting the time requirements. To optimize the use of end devices' computing resources, a larger share of computing tasks should ideally be allocated to them. Yet, to maintain a high overall success rate, only a limited number of frames should be assigned. Therefore, striking a critical balance between reducing the load on the edge server and achieving a high success rate is essential.

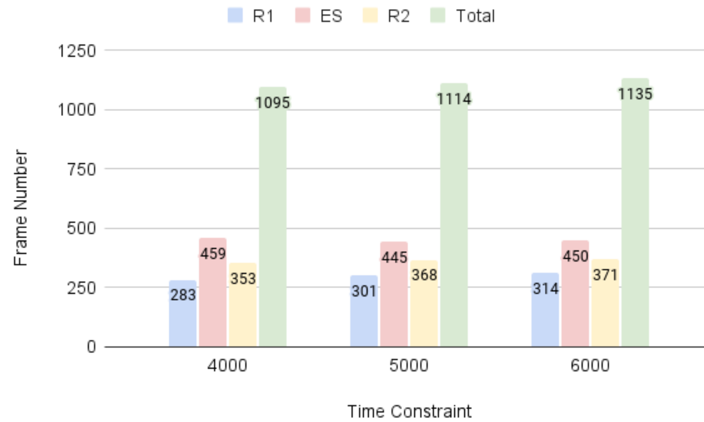


Figure 4.14: 10 Percentage Increase

A key factor impacting the success rate is the timing of the scheduling decisions. For instance, when the scheduler decides to allocate a process locally, it bases its decision on the profile for M containers that are currently running. However, as the frame queue grows and more containers become available, the number of active containers may increase up to the total capacity of N

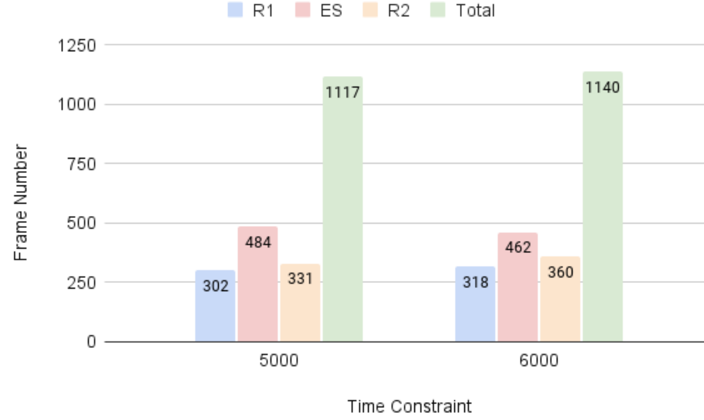


Figure 4.15: 30 Percentage Increase

containers. This difference in timing between the initial decision and the actual processing can adversely affect the final performance. To enhance the overall success rate, an adjustment to the profiling time on end devices can be made using the following formula:

$$T'_m = \alpha(T_n - T_m)(T'_m < T) \tag{4.3}$$

T_n represents the actual evaluation running time for a new container alongside N other active containers, while T_m denotes the actual evaluation running time for the new container operating with M containers. The adjusted time, T'_m, is then recorded in the profile. The variable α reflects a configurable percentage; a higher α results in reduced load on the edge server but potentially a lower success rate. Conversely, a lower α implies a greater load on the edge server, which may enhance the success rate. The time constraint is denoted by T and T'_m must be less than T, the given time constraint, for frame allocation to occur. Otherwise, adjusting T'_m becomes inconsequential as no frames would be allocated.

Table 4.3 presents the actual evaluation time for a Raspberry Pi as detailed in Table 5.2, alongside the adjusted times when α is set to 10%, 30%, and 50% increase. For instance, with α at 10%, the calculation $T_1 = (8744 - 3055) \times 10\%$, results in an adjusted running time of 3626ms for the new container, assuming no other containers are active.

Figures 4.14, 4.15, and 4.16 show results for α at 10%, 30%, and 50%. With α at 10%,

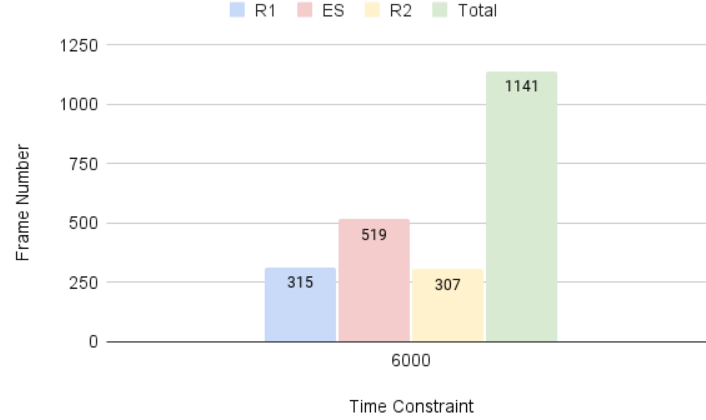


Figure 4.16: 50 Percentage Increase

adjustments in running time are noted under the condition $T'_m < T$ for time limits of 4000ms, 5000ms, and 6000ms. For α at 30%, times exceed the 4000ms limit, showing results only for 5000ms and 6000ms. At 50%, results are shown solely for 6000ms, adhering to the same principle. The results highlight the following key points:

- The higher α is, the less it reduces the load on the edge server, potentially resulting in a higher success rate. Consider a time constraint of 6000ms for example: when α is set to 10%, 283 out of 331 frames (85.4%) on R1 meet the time constraints, and 353 out of 355 frames (99.4%) on R2 meet the time constraints, resulting in an overall performance of 1095 out of 1145 frames (95.6%). Conversely, when α is increased to 50%, 315 out of 319 frames (98.7%) on R1 meet the time constraints, and all 307 out of 307 frames (100%) on R2 meet the time constraints, leading to an overall performance of 1141 out of 1145 frames (99.6%).
- As the edge network expands, adjusting α according to Equation (4.3) offers a valuable method for balancing the workload on the edge server with overall performance.

4.5 Conclusion

This study introduces a distributed system designed to facilitate AI applications at the edge through parallel computing. To address the challenges posed by device diversity and the fluctuat-

ing computational and network conditions inherent to IoT environments, we introduce a dynamic distributed scheduler that adjusts task allocation in real-time based on current operational states. This scheduler, grounded in comprehensive device profiling, is further refined through two strategies: 1) predictive profiling with dynamic updates during task allocation, and 2) fine-tuning device profiles for enhanced performance. Our experimental findings demonstrate the broad applicability and effectiveness of this dynamic distributed scheduler. For future work, we plan to delve deeper into optimal-level scheduling and integrate it with current research to further enhance scheduling efficiency.

Chapter 5

A Dynamic Distributed Scheduler for DNN Inference on the Edge

5.1 Background

The interaction with mobile devices is evolving as they become more personal and capable. Intelligent Personal Assistants (IPAs) like Apple Siri, Google Now, and Microsoft Cortana are becoming standard features, with their use expected to rise alongside the popularity of wearables and smart home technologies. DNN-based applications for speech or image queries demand significantly more processing power than text inputs, a task that traditional mobile devices struggle to handle efficiently in terms of latency and energy consumption [105]. Historically, the solution has been to offload this computational load to high-end cloud servers [106]. This method, however, involves uploading large data volumes over wireless networks, incurring high latency and energy costs. To meet heavy requirements and fully take advantage computing resources on edge nodes, a more sophisticated way is to partition the DNN model and distribute across end computing nodes, the edge server, and the cloud.

Applications of DNNs span various domains: Computer Vision (CV): Here, DNNs are deployed to identify and extract image features, subsequently classifying the images into designated categories. Automatic Speech Recognition (ASR): In this field, DNNs process speech feature vectors, which undergo further post-processing to yield precise text transcriptions. Natural Language Processing (NLP): DNNs in NLP are used to delve into and elucidate the semantic and syntactic nuances of word embedding vectors derived from textual inputs.

Horizontal partitioning of the Deep Learning (DL) model, distributing it across the end, edge,

and cloud layers, is the most prevalent method of segmentation. An alternative approach to model segmentation involves vertical partitioning. Unlike horizontal partitioning, which splits the model layer by layer, vertical partitioning segments the layers into a grid pattern. This method effectively divides the CNN layers into separate computation tasks that can be distributed independently.

Once the initial layers are processed, the intermediate results tend to be significantly smaller in size. This reduction in data size makes it quicker and more efficient to transmit these results over a network to an edge server[84]. Considering the layer's characteristics, this thesis adopts horizontal partitioning to distribute DNN layers along the end, edge, and cloud.

5.2 Related Work

According to the survey paper [107], typical partition constraints encompass factors such as accuracy, energy, latency, network conditions, and privacy. The granularity of partitioning is commonly classified into several types: layer partitioning, data partitioning, sub-layer partitioning, and DNN tuning. Current research primarily targets optimization objectives aimed at minimizing latency, memory usage, and energy consumption, or maximizing accuracy, privacy, and throughput. Popular optimization techniques include dynamic programming, integer programming, machine learning, and early exiting, among others.

Table 5.1 summarizes current research on DNN partitioning, including research objectives, methods, advantages, and disadvantages. Refer to Chapter 2 for more reference details. The thesis aims to optimize DNN inference scheduling by maximizing task fulfillment while utilizing all edge computing resources effectively. However, current research reveals several shortcomings: 1) Lack of consideration for crucial constraints such as latency, energy, and privacy [78, 79]; 2) Insufficient flexibility for dynamic edge environments [80]; 3) Inadequacy in distributing DNN applications across end devices, edge devices, and the cloud [83, 84, 87]; 4) Ineffectiveness in determining partition points in architectures with multiple end devices [75].

Table 5.1: Literature review on DNN Partition

Ref.	Objective	Dynamic	Method	advantage	Disadvantages
[78]	Energy	Yes	Offloads a thread	Used to manage multi-threaded application	Without considering enough variables like network.
[79]	Latency	Yes	enables interactive perception applications on mobile devices	Lightweight online profiler, simple execution time predictor	Without considering the entire application
[80]	Energy	No	Combination of static analysis and dynamic profiling	Energy Efficient	Not flexible
[81]	Energy	Yes	Dynamically determine which parts of a mobile application can be executed remotely	Energy Efficiency, Improved Performance	Need continuous profiling and dynamic adjustment
[75]	Energy, Latency	Yes	ML	Considering multiple constraints	Only one partition point
[82]	Latency	Yes	Early Exit	adapted to complex network conditions	heavy-weight framework
[86]	Energy	Yes	Lyapunov optimization	Multiple end devices, Different network condition	Not parallel computing
[83]	Energy, Memory	Yes	MapReduce, Sharing	Parallel computing on multiple devices	Vertically partitioning
[84]	Memory	Yes	Traversal, Sub-Layers tuning, Sharing	Lightweight framework, Reduxe communication and task migration overhead	Vertically partitioning
[87]	Memory	Yes	Model compression, Distributed inference	Communication aware	Without considering heterogeneous devices

5.3 Scheduler Design

Deep Neural Networks (DNNs) are structured as directed graphs, where each node functions as a neuron processing incoming data to produce outputs. An example depicted in 5.1 shows a 5-layer DNN used for image classification, with computational flow directed from left to right. The connections between neurons map the data flow, forming layers of neurons that perform identical functions on different input segments. During a DNN's forward pass, the output from one layer serves as the input for the next, with the network's depth determined by its total number of layers.

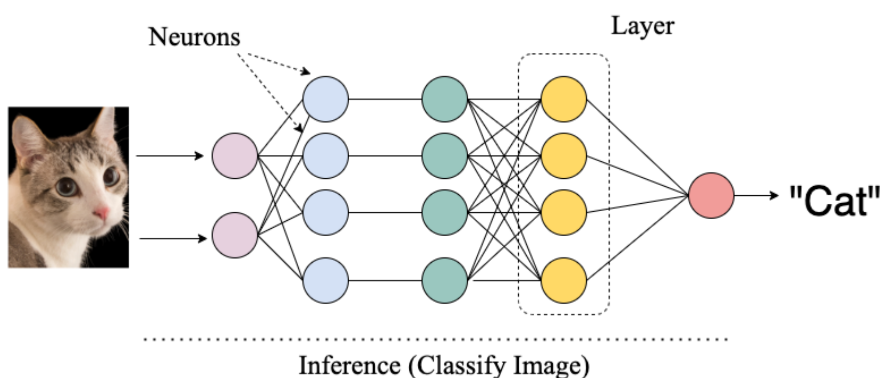


Figure 5.1: A 5-layer DNN Classifies Inference

For applications like face detection, which are relatively lightweight, the task can be processed entirely locally or offloaded to an edge server. However, for DNN models, a fractional offloading approach is more feasible in edge environments with limited computing resources. In this partitioning method, some layers are processed on end devices while others are handled by the edge server, as demonstrated in Fig. 5.2. This example of horizontal partitioning shows a 5-layer DNN model where node 1 processes the first two layers locally before sending the results to the edge server, which may further distribute tasks to other nodes like node 2.

Compared to full application offloading, the main challenge in partial offloading is identifying the optimal points for partitioning the layers. This requires a careful balance between minimizing latency and maximizing computational efficiency across devices. This paper proposes a design of a dynamic distributed scheduler for performance improvement.

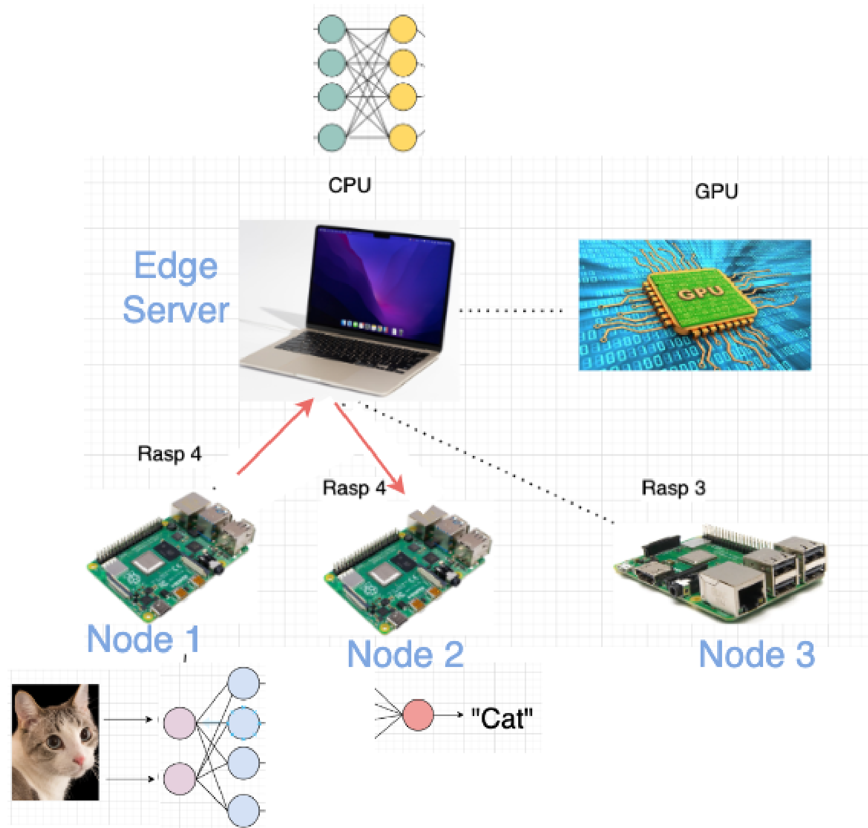


Figure 5.2: A DNN Inference Distribution Example

5.3.1 Scheduler Design Principles

Edge IoT environments are characterized by their dynamic and hybrid nature, encompassing a diverse spectrum of devices from resource-constrained units to robust edge servers. The variability in communication and computational demands, driven by real-time events, necessitates that schedulers dynamically adjust to these changes during runtime. However, this requirement for dynamic adaptation mandates continuous monitoring of current computing and networking conditions, which introduces additional performance overhead. Thus, achieving a balance between this overhead and the performance improvements from near-optimal scheduling—enabled by accurate, real-time updates on compute and network states—is crucial for effective task scheduling and enhanced overall application performance.

In the context of DNN models, with global information, the distributed scheduling coordinator

makes near-optimal decisions based on the local compute principle at the end devices. This approach prioritizes executing tasks on the end device possessing the necessary data, provided it can satisfy all application constraints such as real-time latency requirements. This principle ensures tasks are processed on the device managing the IoT sensor that requires the data. Conversely, if an end device cannot meet all application constraints through local execution, the task may be entirely or partially offloaded to other end devices or the edge server, ensuring efficient and timely task execution.

5.3.2 Device Profiling

Central to our scheduling methodology is the role of edge server scheduling, which is crucial for optimizing overall system performance. The edge server not only determines which device will execute a task but also assesses the computational capacity of the end device, specifically how many layers it can effectively process.

To address these challenges, we propose a proactive solution based on the pre-evaluation of sub-models. This approach involves estimating the processing time between successive layers on each device. For instance, we might calculate the time required for a sub-model to execute from Pooling 1 to Pooling 2 in VGG models. This information enables the edge server to make informed scheduling decisions that comply with predefined time constraints.

However, a significant challenge arises from the impracticality of obtaining processing times for all sub-model configurations. We will explore this issue further and discuss our strategies for overcoming it in Section 5.4.

5.3.3 Architecture Workflow

When an end device such as a Raspberry Pi (R1) receives a classification request, it forwards this request to the edge server. As all end devices regularly update their profiles with detailed processing and transmission times, the edge server leverages this comprehensive global information to make optimal scheduling decisions and provide timely responses.

Algorithm 2 The Dynamic Distributed Scheduler Algorithm

Input: profile tables of all computing nodes, time constraint T

Output: number of frames meeting time constraint, video with augmented reality effect

```
1: N                                        ▷ total layer number of the DNN model
2: R_ab                                     ▷ time of sub-model from layer a to b on Rasp
3: E_ij                                     ▷ running of layer i to j on the edge server
4: G_mnL                                    ▷ time of layer m to n on GPU with workload L
5: T_RG                                     ▷ transmission from Rasp to GPU
6: T_GE                                     ▷ transmission from GPU to edge server
7: if R_0N < T then
8:     processing locally on Raspberry Pi
9: else
10:    for do each b in N...1
11:        if R_ab + G_bNL + T_RG < T then
12:            layer a b on Rasp, b + 1 N on GPU
13:        else
14:            if R_ab + G_bjL + E_jN + T_RG + T_G < T then
15:                layer a b on Rasp, b + 1 j on GPU, j + 1 N on edge server
16:            end if
17:        end if
18:    end for
19: end if
```

Table 5.2: DNN System Configuration

Component Name	Specifications
Edge Server	2.3GHz Dual-Core Intel Core i5, 8 GB RAM, 256GB Disk
Raspberry Pi	Quad core Cortex-A72 (ARM v8) 64-bit 8GB RAM, 1.8GHz Clock Speed
GPU	128-Core Maxwell GPU, Quad-Core ARM @ 1.43GHz CPU, 4GB RAM

The system configuration, comprising a Raspberry Pi (R1), a GPU, and an edge server, is detailed in Table 5.2, and the corresponding scheduling algorithm is outlined in Algorithm 2. If R1 is capable of processing the entire inference task locally, it proceeds without further coordination. If, however, R1 cannot meet the latency requirements independently, the scheduler determines an optimal cutting point to maximize the layers processed on R1 while adhering to time constraints. The GPU, generally more powerful than R1, is tasked with processing the subsequent layers. In this setup, the edge server functions primarily as a coordinator for scheduling rather than as a computational resource for DNN tasks. Nonetheless, in scenarios where the GPU reaches capacity, it may partition the model further, delegating the processing of additional layers to the edge server to maintain efficiency.

In light of the algorithm, the scheduling overhead can become significant for DNN models composed of numerous layers. Pooling layers, however, represent an advantageous cutting point, as they significantly reduce both processing time and output data size, thereby minimizing data volume and enhancing processing speed. Consequently, we advocate for the use of pooling layers as strategic cutting points. Detailed discussions and justifications for this approach will be elaborated in Section 5.4.

5.4 DNN Model Evaluation

we propose to solve the DNN partition problem in the following steps 1) Characterizing Layers in DNN Models and 2)Pre-evaluation of DNN Model.

5.4.1 Characterizing Layers in DNN Models

The various types of layers in today's DNN models include:

(1) Fully-connected Layer (fc)- Every neuron in a fully connected layer is linked to all neurons in its previous layer. This layer calculates the weighted sum of the inputs using predetermined weights

(2) Convolution & Local Layer (conv, local) - In convolution and local layers, an image is processed through convolution with learned filters to generate feature maps. The variation among these layers comes from the size of their input feature maps, the dimensions and quantity of the filters, and the stride at which these filters are applied.

(3) Pooling Layer (pool) - Pooling layers group features together by applying a predefined function, such as max or average, to regions of input feature maps. The key differences in these layers lie in the size of their input, the dimensions of the pooling region, and the stride at which pooling is executed.

(4) Activation Layer (act) - Activation layers introduce non-linearity to neural networks by applying specific functions to each input individually, outputting data in equal volume.

Other layer types include normalization layer(norm), softmax layer (softmax), argmax layer (argmax),dropout layer (dropout), etc.

Fully-connected layers introduce high latency, while convolution and pooling layers at the network's front end show shorter latency. As convolution layers increase and pooling layers decrease data size, this reduction progresses toward the back end where fully-connected layers are situated. This dynamic presents a unique opportunity for computational partitioning in the middle of the DNN, optimizing between the mobile and cloud environments.

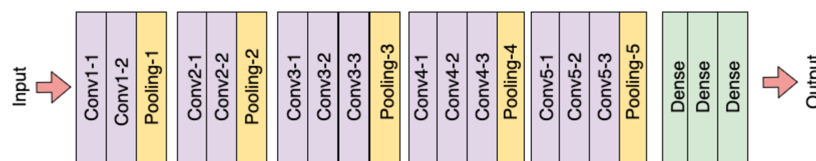


Figure 5.3: VGG16 Architecture

Take VGG16 as an example [108]. VGG16 is a convolutional neural network designed for object detection and classification, achieving an accuracy of 92.7%. It consists of thirteen convolutional layers, five Max Pooling layers, and three Dense layers, totaling 21 layers, as Figure 5.3 shows. However, only sixteen of these layers contain learnable parameters. VGG16 accepts input

tensors of size 224 by 224, each with 3 RGB channels.

Table 5.3: Output Data Size at Each Layer

Layer	input1	conv1-1	conv1-2	pool1	conv2-1	conv2-2	pool2	...
Data Size (MB)	0.57	12.25	12.25	3.06	6.13	6.13	1.53	...

5.4.2 Pre-evaluation of DNN Model

We propose to evaluate sub-model-based profiling at each pooling layer, which includes

(1) Data Size Variations - Evaluate the output data size for each pooling layer on a specific device. In neural networks, convolution layers initially expand data which is subsequently reduced by pooling layers, progressively diminishing the data size through the network. As a result, the data size in the final layers is smaller than the original input.

(2) sub-model Latency - Evaluate the sub-model running time at each pooling layer, as observations indicate that incorporating a pooling layer typically results in reduced processing times for sub-models.

A shorter processing time and reduced output data size make a pooling layer an ideal cutting point, as it both minimizes data volume and accelerates processing. Additional evidence will be provided to support this assertion.

We designated a 1.9MB image as the input and observed the data size changes across each layer, with a subset of the results presented in Table 5.3. The initial layers involve a pattern where convolutional layers increase data volume, followed by pooling layers that decrease it, effectively reducing the overall data size progressively. For instance, the data size at the input layer stands at 0.57MB, which escalates to 12.25MB post-Conv1. This increase is attributed to the Conv-1 layer's utilization of 64 filters, while subsequent layers—Conv-2 with 128 filters, Conv-3 with 256 filters, and Conv-4 and Conv-5 each deploying 512 filters—substantially enhance the depth relative to the

original input. Each pooling operation, employing a 2×2 kernel with a stride of 2, diminishes the height and width of the feature maps, exemplified by Pooling1 which reduces the data size from 12.25MB to 3.06MB.

To fully leverage the computing resources of edge devices, it is crucial to balance data transfer times with computation latency. Therefore, we evaluated the sub-model latency on an edge server, with system specifications detailed in Table 5.2, and part of results are shown in Table 5.4. Sub-model latency measures the processing time from the input layer to a given layer, focusing particularly on the last convolution layer of each block and the subsequent pooling layer. Our findings indicate that deeper layers generally exhibit increased processing times. Interestingly, we observed that the inclusion of a pooling layer typically reduces running time. For instance, the running time after the first convolution layer (Conv1-2) is 97.33 ms, while the subsequent pooling layer (Pooling1) decreases to 96.73 ms, with later blocks following this trend.

Table 5.4: Sub-Model Latency

Layer	Conv1-2	pool1	Conv2-2	pool2	Conv3-3	pool3	Conv4-3	pool4	Conv5-3	pool5
Time (ms)	97.33	96.73	145.27	143.10	237.71	224.46	294.06	291.06	319.16	318.02

Adding a pooling layer typically reduces sub-model processing time due to two factors: 1) Data Movement and Copying: Splitting a model introduces overhead from data transfer between hardware components or within the GPU, which can negate computational savings from fewer operations due to pooling. 2) Model Caching and Optimization: Deep learning frameworks optimize computational graphs more effectively when models are run as a whole rather than in parts. These optimizations include layer operations and memory usage across the entire model, and splitting the model can disrupt this. Thus, a pooling layer serves as an effective cutting point because it decreases data size and enhances processing efficiency.

5.5 Evaluation

We established various scenarios to assess the effectiveness of the DNN layer partitioning and dynamic scheduling algorithm. We began with Case 1, using a Raspberry Pi as the edge device and an edge server, and progressed to Case 2, which incorporates GPU utilization.

5.5.1 Case1: Raspberry Pi and Edge Server

In this scenario, the Raspberry Pi receives requests for 50 VGG16 inferences and forwards them to the edge server. The edge server, serving as the coordinator with comprehensive global information, makes scheduling decisions for each inference. It determines whether to split the model and at which point to execute the split, then communicates these decisions back to the Raspberry Pi. If the model is split, the latter half will be offloaded to the edge server for further processing.

We first determined the success rate, defined as the proportion of inferences completed within the time constraints. The results, presented in Table 5.5, show that our dynamic distribution scheduler achieved a success rate of 98%. This significantly outperforms the Neurosurgeon algorithm, which achieved a success rate of 82%, underscoring the superior efficiency of our scheduler.

Table 5.5: Success Rate

Algorithm	Dynamic Scheduler	Neurosurgeon
Success Rate	98%	82%

We also assessed the edge device utilization rate, which reflects the proportion of layers processed on the endpoint device. The findings, detailed in Table 5.6, reveal that the utilization rate for our dynamic distributed scheduler stands at 33%, which is 10 times higher than that achieved by the Neurosurgeon algorithm.

Table 5.6: Edge Device Utilization Rate

Algorithm	Dynamic Scheduler	Neurosurgeon
Success Rate	33%	4%

5.5.2 Case2.1: Rasp and GPU

GPUs excel in handling computer graphics and image processing due to their highly parallel architecture, making them superior to general-purpose CPUs for algorithms that process large data blocks simultaneously. Recently, GPUs have become increasingly popular for demanding computational tasks in areas like deep learning and data mining.

We incorporated a GPU to highlight the effectiveness of our dynamic distributed scheduler in this scenario. The Raspberry Pi receives requests for 50 DNN inferences and forwards them to the edge server. Acting as the coordinator with comprehensive global information, the edge server makes scheduling decisions for each inference, including whether to split the model and at which layer to execute the split, and then relays these decisions back to the Raspberry Pi. If the model is split, the GPU instead of the edge server functions as an additional processing node.

5.5.3 Case2.2: Rasp, GPU, and Edge Server

This scenario mirrors case 2.1, with the distinction that the edge server also functions as a computing node. If the GPU is overloaded and unable to handle the current task, the model may either be offloaded from the GPU to the edge server or the GPU may split the model at a specific layer, offloading the subsequent layers to the edge server for processing. All decisions are made by the edge server when the Raspberry Pi sends a request to it.

Figure 5.4 presents the experimental results for scenarios 2.1 and 2.2, under varying GPU loads ranging from 0ms to 100ms. As the GPU load increases, the DNN processing capacity diminishes. The blue line represents the system's success rate when the edge server functions as a computing node, while the red line depicts the success rate when the edge server solely acts as a scheduling

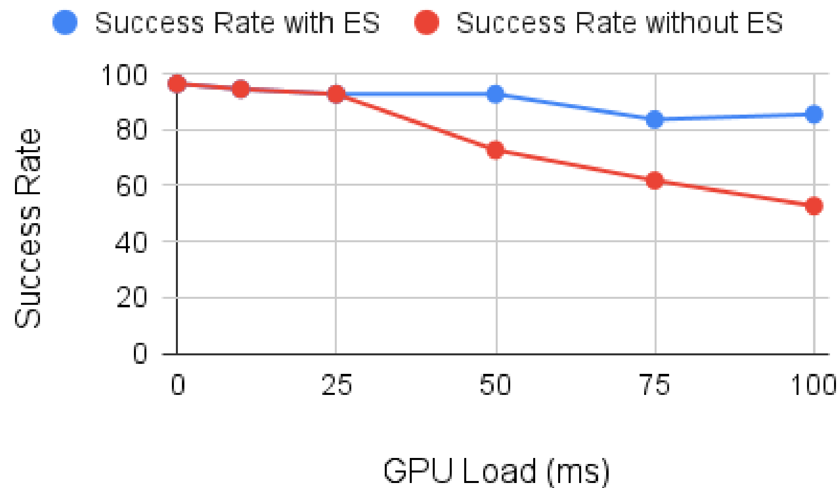


Figure 5.4: Success Rate with Variable GPU Load

coordinator. As illustrated by the blue line, an increase in GPU load correlates with a decrease in success rate. This decline indicates that when the GPU is burdened with additional tasks, its ability to handle parallel computations is compromised, leading to longer processing times and lower success rates. The red line exhibits a similar pattern. A comparison between the two lines reveals that as the GPU load escalates, the edge server assumes some tasks that the overloaded GPU cannot process promptly, thereby fulfilling the time constraints.

5.6 Conclusion

this paper introduces a dynamic distributed scheduler optimized for DNN inferences in edge computing environments. Our approach diverges from traditional static scheduling by adapting in real-time to changes in computing and network states, thus enhancing resource utilization. Key contributions of this work include 1) A novel model partitioning strategy that segments DNN models into multiple sub-models, allowing for more flexible and efficient use of edge computing resources. 2) The introduction of sub-model profiling, a pioneering method that optimizes DNN layer partitioning to reduce pre-evaluation work and adapt to various architectures. 3) A prototype implementation of the scheduler integrated with a GPU, extensively tested to demonstrate superior

performance over existing edge scheduling solutions. This research enhances the capabilities of edge computing frameworks to meet the stringent requirements of modern IoT applications, establishing a foundation for future advancements in distributed computing.

Chapter 6

Middleware Evaluation

Edge computing is envisioned to be the driving technology for building highly sophisticated context aware applications for IoT and smart city environments. It pushes the frontier of computing applications, data, and services away from centralized nodes to the logical extremes of a network and thereby addresses two critical problems incurred in cloud computing, namely high latency and high data transfer cost.

At present, there is a large gap between the IoT application developer requirements and the underlying IoT infrastructures. On one hand, IoT application developers need sophisticated system support services such as resource discovery, sharing of peripheral IoT devices, and powerful computing and communication abstractions to develop context-aware applications. On the other hand, current IoT devices and supporting software infrastructures have mostly been developed in silos with little or no portability or opportunity for code reuse. The goal of an edge computing middleware layer is to bridge this gap. Indeed, a major effort has been going on in this direction both in academia and industry, and several middleware layers for edge computing have been proposed. Examples include EdgeX, Kura, IoT Edge, OpenEdge, etc..

These middleware layers are based on a microservice-based architecture consisting of a suite of small services, each running in its isolated computing environment such as a container, and communicating with other services using an appropriate intercontainer communication mechanism. An IoT application is then constructed by appropriately composing these different microservices. A middleware layer for edge computing then provides a set of system-level microservices that can be

used to build context-aware distributed applications at the edge. The main goal of this paper is to evaluate some of these prominent middleware layers to gain an understanding of the nature of system abstractions they provide, the generality in terms of the range of diverse IoT applications that can be built using these layers, the ease of use to construct such applications, and their performance implications on the overall performance of those applications. Since a middleware layer is a software layer between an application and the underlying IoT infrastructure, it is expected to introduce some performance overhead. Through this evaluation exercise, we seek to understand what the right tradeoffs would be between the level of support they provide to the application developers and the performance overhead they introduce.

In particular, we evaluate two popular edge computing middleware layers, EdgeX Foundry and Kura. EdgeX Foundry is a vendor-neutral open-source platform hosted by the Linux Foundation, providing a common framework for industrial IoT edge computing. Eclipse Kura is an extensible open-source IoT Edge Framework based on Java/OSGi. Kura offers API access to the hardware interfaces of IoT Gateways (serial ports, GPS, watchdog, GPIOs, I2C, etc.). To evaluate the generality of the middleware layers, we have experimented with application scenarios ranging from simple, straight-forward ones where an edge server simply receives sensor data and publishes it on a cloud server to complex application scenarios involving multiple sensors, actuators, complex data processing and interactions with users.

An important aspect of our evaluation is that we have experimented with IoT application scenarios that involve humans in the loop, both as data generators and data consumers. This is an important feature of sophisticated IoT applications that is often overlooked in the design of middleware layers for edge computing.

This chapter makes the following important contributions:

- To the best of our knowledge, this is the first extensive evaluation of middle layers for edge computing encompassing a range of diverse IoT applications and application architectures.
- The evaluation includes IoT application scenarios that involve humans in the loop, an aspect

that has been mostly overlooked so far.

- The paper provides a set of very important insights based on the evaluation that will help advance the design and development of middleware layers for edge computing.

6.1 IoT Architectures and Scenarios

6.1.1 Middleware Architectures

For our edge middleware layers evaluation, we have experimented with three different architectures: the baseline architecture, the EdgeX-based architecture, and the Kura-based architecture. These architectures differ from one another in how the data flows within the edge server.

The baseline architecture is a minimal and bare-bone architecture. It has just enough functionality to be able to run the applications that we have experimented with. These applications are described in the next section. The EdgeX-based architecture on the other hand is the most complex and feature-rich architecture. It has several sophisticated features such as logging and security.

Current IoT environments are dominated by M2M (machine-to-machine) communication. An administrator sets up and configures the system with a certain data flow in mind, i.e., collects the data from a sensor or a number of sensors, summarizes/analyzes the data, and then publish it to the cloud or some enterprise server. However, emerging IoT applications involve humans in the loop, both as data generators and data consumers. To address this, we have experimented with scenarios that involve humans affecting the data flow in addition to the traditional IoT setups with M2M communication. For this reason, we have included an interface server component in the edge server that enables human user interaction with the edge middleware.

The interface server

The interface server is the entry point to the system. It is a service built using the Go programming language. For applications that involve humans in the loop, this server handles all communications with end users, and for applications without humans in the loop, this server ini-

tiates the application requests. It is started with application-dependent parameters and supports multiple clients.

Baseline Architecture

The baseline architecture illustrated in Figure 6.1 includes a single service, built using the Go programming language. This service receives requests from the interface server, interacts with the Raspberry Pi requesting and receiving captured images, and finally forwarding these images to the appropriate application. This service is run inside a docker container. The images are passed through the use of shared volumes between the containers.

As shown in Figure 6.1, Docker Container acts as the edge server for users, Raspberry Pi, and the cloud. It receives service requests such as face detection from the users and provides response back to them. It interacts with Raspberry Pi to retrieve and send the sensor data, e.g. interacts with the camera to snap and send the current picture. With this data, it also starts the appropriate container(s) to run the requested service, e.g. detect faces in the picture. After processing, the data is directly sent back to the user or sent to the cloud depending on the application.

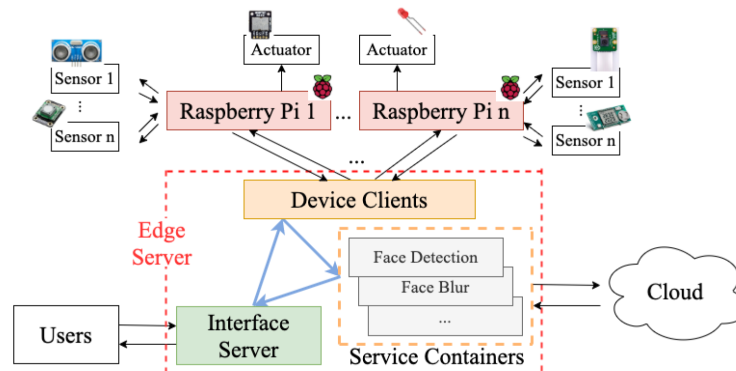


Figure 6.1: Baseline Architecture

EdgeX Based Architecture

EdgeX Foundry is a collection of open-source microservices that is deployed from the Device Services Layer at the edge to the Core Services Layer at the center, allowing users to run it in Docker

containers. Application Services are designed to process data from Edgex Foundry to external systems. It also provides a means to send it to the endpoint. Device Services are edge connectors for IoT objects like sensors, actuators, and so on.

As shown in 6.2, Edgex Foundry acts as the edge server to interact with users, Raspberry Pi, and the cloud. It receives service requests from users and interacts with Raspberry Pi to get sensor data, e.g. the current picture from one camera. As Edgex has inside Application Services, the picture is sent to them to get processing like face detection. The processed data is sent back to the user or the cloud for further execution.

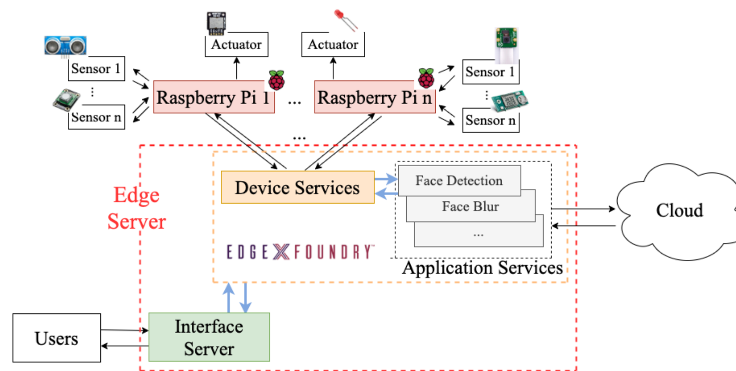


Figure 6.2: Edgex Based Middleware Architecture

<u>Kura Based Architecture</u>

Eclipse Kura aims to provide Java/OSGi-based containers for M2M (machine-to-machine) applications running in IoT gateways. Kura components are designed as configurable OSGi declarative services that expose API and raise events. So Kura achieves the separation of the actual operating environment and the development environment. The function written in the development environment is an OSGi bundle that can be deployed to the operating environment remotely through a plug-in in Eclipse, which improves programming efficiency and also makes IoT functions modular and structure clearer.

In the development environment, the Eclipse plug-in project is a writable OSGi bundle that achieves a high degree of modularity, and this plug-in project can be exported as a JAR package that is imported to the operating environment. The JAR package can be reused in the development

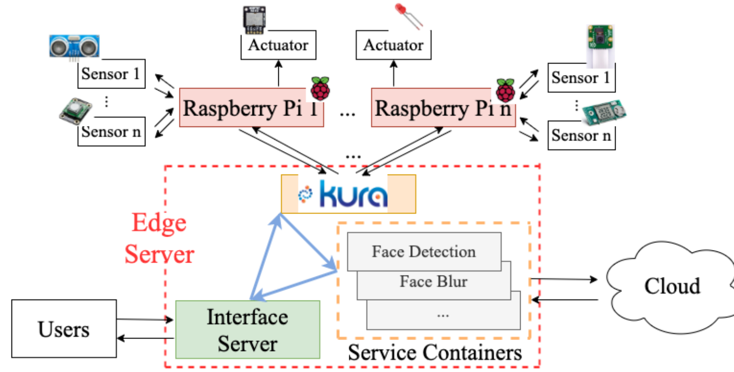


Figure 6.3: Kura Based Middleware Architecture

environment via the Kura emulator. In this paper, we used the latter method to run the server service in Eclipse Kura.

As shown in Figure 6.3, Kura acts as the edge server for users, Raspberry Pi, and the cloud. The service part of it is similar to the baseline architecture. When Kura receives a request from a user, it interacts with Raspberry Pi to get sensor data, e.g. the current picture from one camera. It then sends the picture to the service containers for processing. The processed data is sent back to the user or the cloud for further execution.

6.1.2 Edge-IoT Scenarios

Edge computing and IoT can be used and deployed in many scenarios. Here we highlight the scenarios/setups that we considered in our evaluation.

Fully-automated Scenarios

The *Fully-automated Scenarios* represent the typical IoT usage scenarios covering a large class of IoT applications where an edge server acts as a gateway to send sensor data to more powerful servers, usually residing in the cloud for further processing and storage. This setup is normally fully automated with no human interaction aside from the initial configuration.

Two types of setups fall under this scenario: simple and elaborate. The simple setup is

comprised of a single sensor and an edge server controlling the operation of that sensor. The edge server requests the sensor to send data either on demand or periodically. The sensor in our setup is a camera module connected to a Raspberry Pi. It either sends a single picture or a sequence of pictures snapped periodically. In the first scenario (*Scenario A*), the edge server publishes these snapped pictures to the cloud, while in the second scenario (*Scenario B*), the edge server first blurs all faces in the snapped picture(s) and then publishes them on the cloud.

In the more elaborate automated setup, multiple sensors and actuators are introduced. The input from one sensor causes the edge server to fetch data from other sensors. This data is processed, and based on the result some actuators are triggered. This type of setup is common in industrial IoT scenarios as well as other scenarios where automated action in required.

For the elaborate setup, we have a proximity sensor and a camera as sensors, and an LED as an actuator. In this scenario (*Scenario C*), when the proximity sensor is triggered, it sends a message to the edge server which instructs the camera to snap a picture and send it. The edge server then counts the number of faces in the picture and sends the result to the Raspberry Pi with an LED. The LED is lit if the number of faces exceeds a certain configurable threshold.

Note that this setup with the two sensors and an actuator is a representative of large number of applications involving multiple sensors and actuators. One practical example of this type is an IoT environment with temperature/humidity sensors and fans. Based on the data sent by the sensors, an edge server will control the speeds and the number of fans operating. Another example would be a gate/door with a proximity sensor and a camera. When a person or a vehicle approaches the gate, a picture is snapped by the camera, processed by the edge server, and the gate is open based on the face of the person or the license plate of the vehicle. Performance measurement for these three application scenarios includes end-to-end latency and steady-state streaming throughput.

Human-in-the-loop scenarios

The *Human-in-the-loop scenarios* explore situations where external (non-administrator) users interact and utilize the IoT environment. They expand the fully automated scenarios to add users

that interact with the IoT environment via the edge server. In the first set of application scenarios, a user is a data consumer and is interested in learning something specific about the IoT environment, while in the second set of application scenarios, a user is a data generator as well.

For these human-in-the-loop scenarios, the setup is comprised of a camera that is controlled by an edge server. The edge server provides an interface for external end users (interface server) to utilize its processing power and the camera. In one scenario, (*Scenario D*), the user first sends a request to the edge server to get a count of the number of people in an environment monitored by the camera. The edge server then requests the camera to snap and send a picture, which it processes to count the number of faces. The edge server then returns this number to the user. In another scenario, (*Scenario E*), the user requests a current picture of the environment with the faces of all the people blurred, and the edge server returns the snapped picture to the user after blurring all faces in it. One possible usage for these types of scenarios is in shopping malls or airports where end-users might be interested in knowing the number of free seats in a food court, empty places in a parking lot, or the number of people waiting in a coffee shop line.

In another scenario (*Scenario F*), a user sends a request to the interface server along with some text to check if the text is currently present in the environment monitored by the camera. The edge server then requests the camera to snap and send a picture, which it processes to check if that text is present in the snapped picture. The edge server then returns a Yes/No result to the user. One possible usage for this scenario is shopping malls where users might be interested to check if a certain store sign is there, or parking lots security to check if a certain license plate is there. Performance measurement for these *Human-in-the-loop* scenarios include end-to-end latency measurement.

Scalability scenarios

Finally, the *Scalability scenarios* are an extension to the *Human-in-the-loop scenarios*, as they involve multiple users interacting concurrently with the IoT environment. The goal is to understand how the performance is impacted as the number of users interacting with the IoT environment increases. With multiple users, multiple requests are sent to the edge server simultaneously. The

edge server requests the Raspberry Pi with a camera module to snap and send a picture. Once received, the edge server processes the picture and sends the result back to the users. In Scenario G, the edge server sends a count of the number of faces in the picture, while in Scenario H, it sends the picture with all faces blurred.

6.2 Middleware Evaluation

Any middleware system layer will necessarily introduce some performance overhead to simplify edge programming and enable interoperability and code reuse. Our goal is to quantify this overhead under typical IoT application scenarios.

Our evaluation has been done by developing and experimenting with different applications representing a wide range of IoT application scenarios. Each one of these applications has been implemented and evaluated under three different middleware architectures. Since our goal is to know how each architecture affects the performance of each application, we made sure to use the same code base whenever possible. In our testing, we have used EdgeX v1.1 (Fuji) which uses the Go programming language. So, to eliminate any programming language-related performance differences, we used the Go programming language to build all system components except the Kura-specific service which was built using Java. In addition, following the way EdgeX runs, and to make all system components portable, all of our services and applications run inside docker containers, except the main interface server which runs natively.

6.2.1 Experiment Setup

The experiment setup consists of: 1) an edge server running on a laptop with an Intel Core i7 processor, 16GB of RAM, running Arch Linux, 2) a Samsung Galaxy S8+, used as the client for the single client experiments, 3) a laptop with an Intel Core i5 processor, 8GB of RAM, running Linux to simulate the simultaneous clients requests for the multi-client experiments, and 4) sensors.

The main sensor in our experiments is a Raspberry Pi 4 Model B with a 1.5GHz 64-bit quad-core CPU and 4GB of RAM, with a 1080P 5M pixel camera module connected to it. The Raspberry

Pi runs a Python script that waits/listens for TCP requests from the services running on the edge server. Once a request is received, the camera module is instructed to capture an image, the image is saved and sent to the edge server. For *Scenario C*, a second Raspberry Pi was used with a push button to act as a sensor and an LED to act as an actuator. Two Python scripts were used on this Raspberry Pi, one to react to the button press, and another to light the LED. To measure the performance of the application scenario, the time was measured on this second Raspberry Pi, finding the time difference between the trigger and the actuation.

As we mentioned earlier, aside from the interface server, all components running on the edge server were deployed using docker containers. For the baseline and Kura-based architectures, there were two containers each: one that handles the communications with the interface server and the sensor/Raspberry Pi, and the other one is the application-specific container. The EdgeX-based architecture, on the other hand, uses twelve containers, one of which is the application-specific container.

The application-specific docker containers were built to perform certain tasks and ranged in size and complexity from small and simple to very large and complex. For example, the container for *Scenario A*, whose task is to simply receive an image and publish it to the cloud, has a docker image size of only 8.64MB, while the container for *Scenario F*, which performs text detection and recognition has a docker image size of 411MB.

We built two versions of each application, one using the App Functions SDK to be used with the EdgeX-based architecture, and a stripped-down version to be used with the baseline and the Kura-based architectures. The App Functions SDK is meant to provide all the plumbing necessary for developers to get started in processing/transforming/exporting data out of EdgeX.

Although we have built our prototype to start application containers on demand, the times in our performance evaluation section do not include the startup time for the containers/services, to eliminate any time variations not related to the architecture.

6.2.2 Qualitative Evaluation

Performance is not the sole metric to be considered when evaluating middleware solutions. Ease of use, ease of programming, and customization are also crucial. In developing our prototypes, building the minimal baseline architecture from scratch required considerably less time than customizing/modifying EdgeX to suit our needs. While using the EdgeX framework does not require a deep understanding of the internals of the framework, customizing and modifying it does. Since the EdgeX framework consists of several microservices, understanding the data flow among these microservices was essential and that took a considerable amount of time. On the other hand, adding a new OSGI service to Kura was relatively straightforward. So in terms of use, EdgeX has a significantly high learning curve.

6.2.3 Performance Measurements

To better understand and quantify the effect of introducing a middleware layer to an IoT application at the edge, we measured end-to-end latency for the scenarios described in the previous section under the three different architectures. We also measured the throughput for scenarios A and B. In this subsection, we report the performance numbers measured for different application scenarios under different architectures. In the next section, we provide a detailed explanation of our measurements.

Table 6.1: Fully-automated Scenarios average end-to-end latency

	Real-time Capture					Pre-captured				
	Baseline	EdgeX		Kura		Baseline	EdgeX		Kura	
Scenario A	805ms	820ms	1.80%	797ms	-1.18%	73ms	100ms	37.46%	74ms	1.37%
Scenario B	932ms	946ms	1.45%	937ms	0.51%	223ms	240ms	7.49%	223ms	-0.11%
Scenario C	878ms	948ms	7.97%	916ms	4.33%	201ms	288ms	43.28%	242ms	20.40%

All application scenarios include a step where the sensor captures an image and sends it to the edge server. The size of the captured images can vary based on their content. Therefore, to eliminate any variations in the measurements that might occur from this size difference, we changed the script on the sensor to always send the same image for each run.

For scenarios A through E as well as the scalability scenarios, we captured two pictures, one with four faces (file size 189KB) and another with sixteen faces (file size 208KB). For scenario F, we captured two pictures, one with five words (file size 174KB) and another with sixteen words (file size 181 KB.)

Fully-automated Scenarios

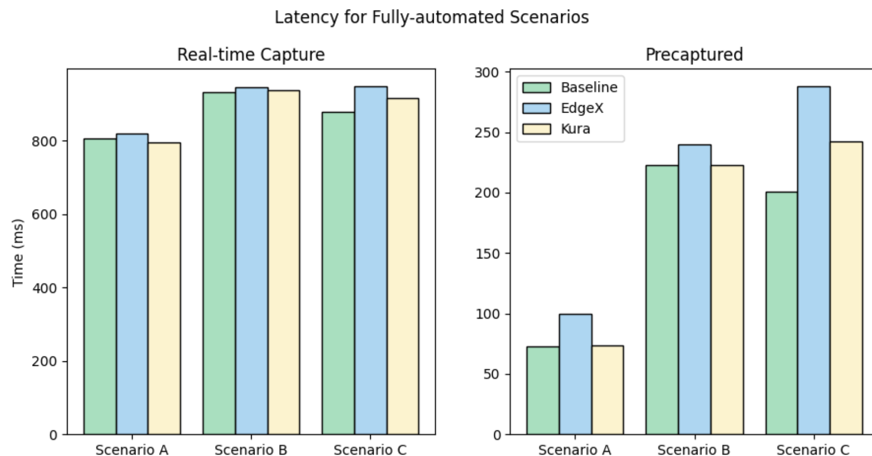


Figure 6.4: End-to-end latency for fully automated scenarios

The average end-to-end latency for all three fully-automated scenarios is reported in Figure 6.4 and Table 6.1 (Real-time capture). For scenario A (publish only), we ran the application ten times, with each run lasting one minute. For each run, the application sends a request to the sensor to capture an image. The latency is the elapsed time measured from the time the request is sent till the time the image is received by the application, i.e. just before the image is published to the cloud. For each one-minute run, the end-to-end latency was measured for each image received and the mean value was computed.

The latency reported is the average of the mean values from ten separate runs. Latency for Scenario B was measured in a similar manner, from the time a request was sent until the time of completion of processing (face detection plus face blur) of the received image.

For Scenario C, the latency is measured as the time duration from the time the proximity sensor is triggered until the time the LED is triggered. Recall that this includes sending a message

Table 6.2: Average Throughput for Fully-automated Scenarios

	Real-time Capture					Pre-captured				
	Baseline	EdgeX		Kura		Baseline	EdgeX		Kura	
Scenario A	74.8	74	-1.07%	76	1.60%	824.4	600	-27.22%	811.2	-1.60%
Scenario B	66	64.4	-2.42%	65.2	-1.21%	268.4	250.4	-6.71%	269.2	0.30%

from Raspberry Pi to the edge server (when the proximity sensor is triggered), the edge server sending a request to the Raspberry Pi with the camera module, receiving an image, performing face counting, and sending the number of faces to the Raspberry Pi with LED sensor, and that Raspberry Pi triggering the LED sensor. To avoid any errors due to differences in the clocks of the three devices, we placed the proximity sensor and the LED actuator on the same Raspberry Pi (where the start and end times were recorded), while the camera module was placed on a different Raspberry Pi.

By examining the latency values in Figure 6.4 and Table 6.1 (real-time capture), we notice that the latency is similar for all three architectures (EdgeX-based architecture has slightly higher latency). This would indicate that the middlewares have only a minimal impact on application latency. We also notice that the latency values are much higher than expected and they cannot be explained by adding the communication and processing delays. We examined each component of the application and the setup and discovered that a large percentage of the latency is related to the camera module. The camera module requires 700ms on average to capture and save an image. This time dwarfs the delays caused by communication and processing, which we expect to be architecture-dependent. Because of this reason, we saw insignificant differences between all three architectures when it came to latency

So, we modified the script on the Raspberry Pi to send pre-captured images without performing a real-time image capture. The updated latency values are shown in the same figure and table under the pre-captured heading.

The throughput for fully-automated scenarios (scenarios A and B) are reported in Figure 6.5 and Table 6.2. To measure throughput, a new request is sent to the sensor immediately after the

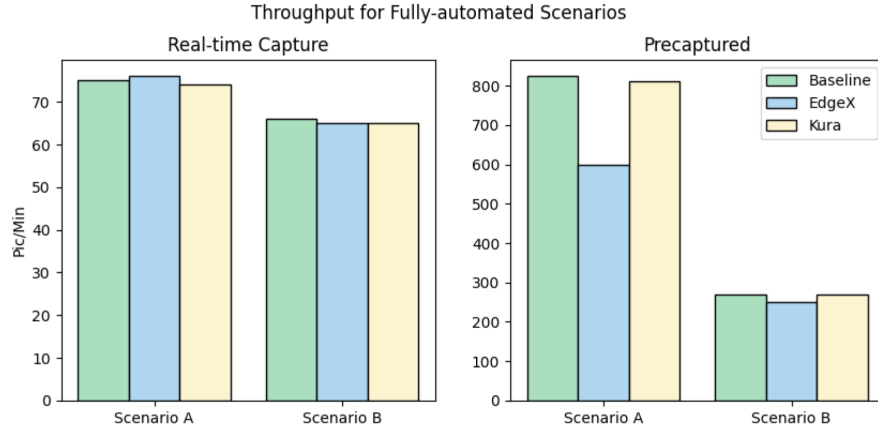


Figure 6.5: Average Throughput for Fully-Automated Scenarios

image from the current request is received by the application, and the throughput is the number of images received in one minute. For five of these runs, we used the image with four faces, while for the other five runs, the image with sixteen faces was used. The throughput reported here is the average computed from ten different runs. As in the case of latency, we observe that the throughput is similar for all three architectures and is dominated by the time the camera module takes to capture and save an image. To address this, we have measured throughput for both scenarios using pre-captured images (reported under the pre-captured heading).

Figure 6.6 and Table 6.3 show the latency measurements for Scenarios D and E for real-time image capture. The latency is measured as the duration from the time a user sends a request until the time that the user receives a reply (face count in Scenario D and the captured image with faces blurred in Scenario E). Once again, we notice that the latency for the three architectures is similar. We attribute this to the time it takes to capture and save images, which dominates.

Human-in-the-loop Scenarios

Table 6.3: Average Latency for Human-in-the-loop Scenarios D and E

	Real-time Capture				
	Baseline	EdgeX		Kura	
Scenario D	965.4	1052.8	9.05%	1026.4	6.32%
Scenario E	1068.67	1093.25	2.30%	1081.5	1.20%

Table 6.4: Average Latency for Human-in-loop Scenario F

	Real-time Capture					Pre-captured				
	Baseline	EdgeX		Kura		Baseline	EdgeX		Kura	
Scenario F	2041.4	2211.3	8.32%	2046.2	0.24%	1284.4	1505.9	17.25%	1380.2	7.46%

Figure 6.6 and Table 6.3 show the latency measurements for Scenarios D and E for real-time image capture. The latency is measured as the duration from the time a user sends a request until the time that the user receives a reply (face count in Scenario D and the captured image with faces blurred in Scenario E). Once again, we notice that the latency for the three architectures is similar. We attribute this to the time it takes to capture and save images, which dominates.

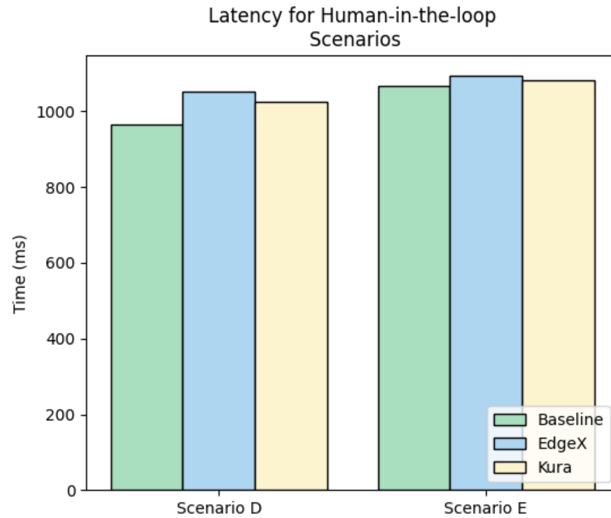


Figure 6.6: Average Latency for Human-in-the-Loop Scenarios D and E

Figure 6.7 and Table 6.4 shows the latency measurement for Scenario F. Recall that in this scenario, there is an increase in the level of interaction as the user acts as a data producer as well as a data consumer. In our implementation, the edge server performs text detection and text recognition using the OpenCV and Tesseract libraries. To determine if the camera module image capture delay will make a difference in the results, we have run this scenario for both real-time image capture and pre-captured images.

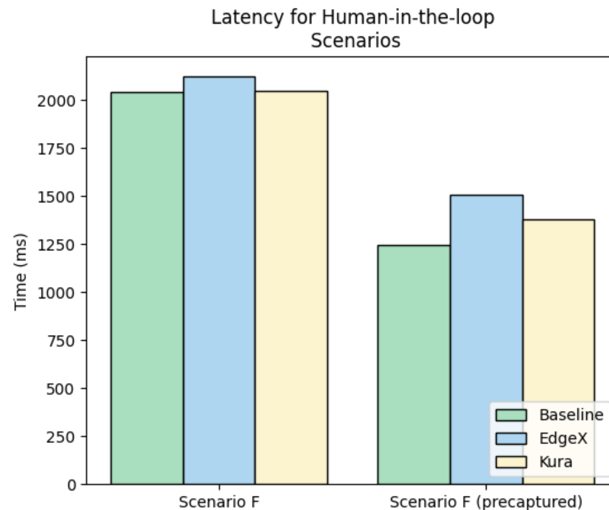


Figure 6.7: Average Latency for Human-in-the-Loop Scenario F

Scalability Scenarios

Figures 6.8 and 6.9 report the latency measurements for the scalability scenarios (Scenarios G and H) for different numbers of concurrent users. Our goal was to assess scalability as the number of concurrent users grows significantly large (up to 100 concurrent users). Using a separate mobile device for each user for this purpose was prohibitively expensive. So, instead, we simulate multiple concurrent users interacting with the IoT environment by running a multi-threaded program that sends multiple concurrent requests and receives replies. This program was run on a different laptop. We experimented with 5, 10, 25, 50, 75, and 100 simulated user connections.

An interesting observation we make here is that in both scenarios, the average latency actually decreases as we increase the number of users until some threshold. The reason is that the interface server may receive multiple requests before the edge server receives the image in response to its first request to the camera module. In this situation, the edge server queues user requests and uses the same result (face count or image with blurred faces) for all the requests received while the image is being fetched and processed. Thus fetching and processing a single image serves multiple users' requests. However, for any user request that is received after the result from the previous request(s) has been sent back, the edge server initiates a new request to the camera module from a new image.

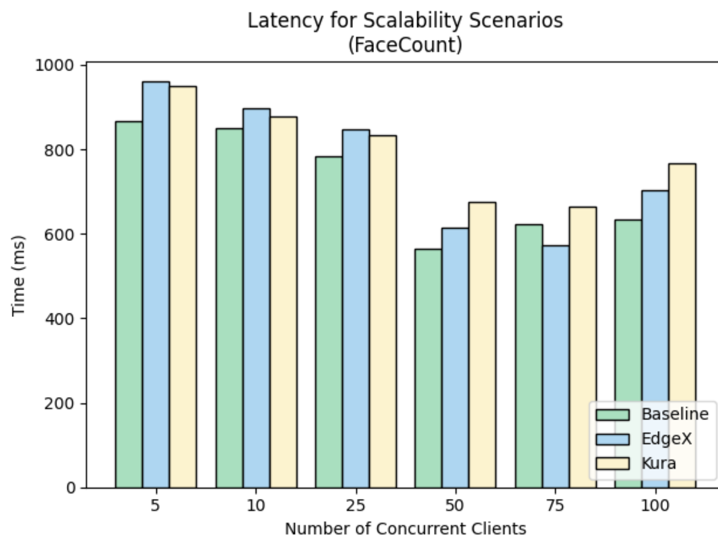


Figure 6.8: Average Latency for Scalability Scenario G

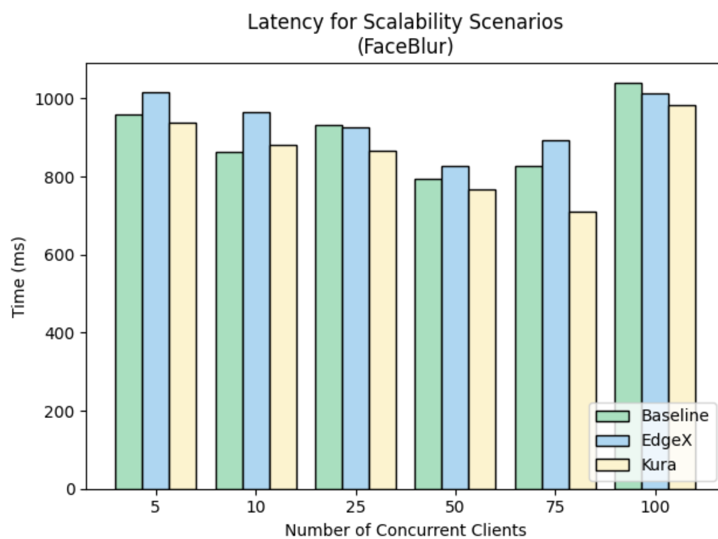


Figure 6.9: Average Latency for Scalability Scenario H

Again to account for the large time to capture and save an image, we ran these scenarios for both real-time image captures and pre-recorded images.

6.3 Discussion

Based on our experimental evaluation of edge middleware layers, we seek to answer three questions:

(1) How does an edge middleware architecture impact the performance of IoT applications?

(2) Can we utilize existing edge middleware architectures that are designed mainly for machine-to-machine interactions to build IoT applications that involve humans in the loop?

(3) What system-level features in an edge middleware architecture would be useful for building efficient IoT applications?

To answer the first question, we examine the results in Figures 6.4, 6.5 and 6.7, and Tables 6.1, 6.2, 6.3, and 6.4. We can see that the choice of the middleware architecture does have an impact on the latency and throughput of the applications, and the magnitude of this impact is dependent on the types of sensor(s) and the characteristics of the application. In our experiments, we noticed high variability in the measurements among different runs. A plausible reason for this variability is that our implementations involve multiple docker containers communicating with one another as well as network connections between multiple devices (laptop(s), Raspberry Pi(s), and a mobile device). Any slight variability in any one of these can have a significant impact on the performance measured. Scenarios A (capture then publish) and B (capture, blur, and then publish) have the least variability as they are the simplest and can be run continuously. For these scenarios, we see that all three middleware architectures have similar performance when sensor data capture time is high (real-time image capture) with differences between the baseline architecture and EdgeX and Kura-based architectures in the range of 0.5%-1.80% for latency and 1.0%-2.5% for throughput. However, when the sensor data capture time is low (pre-captured image), we start to notice a difference in performance between EdgeX and the baseline architectures, while Kura's performance remains similar to the baseline architecture. Latency in EdgeX-based architecture is approximately 37.5% higher for Scenario A and 7.5% higher for Scenario B for pre-captured image cases. Similarly,

throughput in EdgeX-based architecture is approximately 27% lower for Scenario A and 6.7% for Scenario B. For the more complex scenarios (C, D, E, and F), latency in Kura based architecture is generally lower than EdgeX-based architecture and higher than the baseline architecture.

The main finding is that the impact of middleware architecture on application performance is more pronounced when the actual computation time of the application task is relatively high compared to the time needed to capture and transmit sensor data.

The impact of data capture time can be seen when we compare the real-time image capture cases to the pre-captured image cases in Scenarios A, B, C, and F. We can also see the impact of the size of data transfer from an image being transmitted in Scenario E (larger data transmission) compared to a number (the number of faces detected) in Scenario D. Finally, we can see the impact of the difference in processing done on the data from no processing being done in Scenario A vs face counting and face blurring in Scenarios B and C respectively.

To answer the second question, we have shown that it is possible to use existing edge middleware architectures that are mainly designed for machine-to-machine operation to incorporate human users in that interact with an IoT environment. However, the amount of time and effort required to incorporate human users varied significantly. For Kura, it was a matter of creating and adding a new OSGi service, a fairly simple and straightforward task. The new service handles requests sent by our interface server and interacts with the sensor to fetch data. Once received the data is sent to one of several application containers to be processed and then sent back to the interface server which is responsible for forwarding them back to the users.

Incorporating human users in EdgeX-based architecture also required adding new services, but this task was much more involved. EdgeX uses a micro-service design style with each micro-service running as a docker container. To incorporate users, we needed to understand the function of each micro-service, how the data flows from one micro-service container to the next, and how the containers interact with one another. We had to create and add a new device driver service to interact with the sensor, and a new component in our interface server to send the appropriate commands to the appropriate EdgeX micro-service based on the users' requests. We also had to

create and integrate several application micro-service containers to handle/process the data sent by the sensor and forward the results back to the interface server. This was further complicated by the steep learning curve of EdgeX. However, we do expect that once a developer becomes familiar with EdgeX functionality, adding new sensors and applications will become less complex.

Finally, based on our experience with building a variety of applications over a variety of architectures, we have suggestions for several system-level features that would be very useful for developing IoT applications at the edge. Application developers still face many challenges, such as the changing computing environment, enterprise application integration, and so on [**?**]. We believe flexibility and simplicity are two key features. A middleware should be able to handle different setups and environments.

First, to support applications possibly involving both humans in loop and machines in the loop, an edge server needs to provide an interface through which users in the vicinity may discover available services as well as initiate a connection with the edge server as needed. Furthermore, an appropriate authentication mechanism is needed depending on the sensitivity of the service provided.

Another important feature to improve the performance of feature-rich middleware frameworks such as EdgeX is the ability to alter the data path between the containers[109]. For example, a device driver in EdgeX receives data from the sensors, and sends the data to a *core data micro service*, which first inserts the data in a database before sending it to the applications. Inserting the data in a database before sending it to the application introduces performance overhead. Instead, allowing the data to be sent directly to the application would save time and improve performance, and to make sure that the data is eventually inserted into the database, it could be logged in a staging file in a manner similar to a journaling file system.

Another common functionality needed in building complex IoT applications at the edge is intercontainer communication. Any improvement in the efficiency of inter-container communication will go a long way in improving time-sensitive application performance, e.g. see [110]. Finally, some of the container-related issues that the researchers are addressing, such as faster startup times[111] or lighter-weight abstractions[112] will certainly help in application performance improvement.

6.4 Conclusion

This paper evaluates middleware architectures for IoT applications, using a baseline architecture to benchmark two open-source solutions: Eclipse Kura and EdgeX Foundry. Chosen to represent opposite ends of the complexity spectrum—Kura as a minimal solution and EdgeX as a feature-rich alternative—these middleware were tested across various scenarios. Results show that while EdgeX's extensive features introduce some overhead, overall application performance remains largely unaffected in most cases, underscoring the importance of selecting the right middleware for each IoT application.

We also explored modifications to enable external client interactions with EdgeX and Kura. Human-in-the-loop and scalability tests indicate that, while adding this functionality is feasible and valuable for expanding IoT use cases, it can be complex. Lastly, we provide recommendations to enhance system-level support for middleware performance based on our development and experimental findings.

Chapter 7

Conclusion

7.1 Research Summary

This thesis concludes that edge AI systems, primarily designed to address the intensifying demands of AI workloads by reducing latency and processing data closer to its source, represent a significant advancement in the realm of IoT applications. Throughout this work, the dynamic distributed scheduler developed here has proven effective in managing the unique challenges associated with edge computing—particularly in balancing compute-intensive demands with limited edge capabilities.

Our proposed scheduler dynamically adapts to the fluctuating conditions of edge computing environments, intelligently distributing tasks across diverse computing nodes to optimize for latency, privacy, power, and cost constraints. By leveraging detailed device profiling and predictive modeling, it anticipates and adjusts to real-time operational conditions, ensuring that computing resources are utilized efficiently and application performance is maximized.

Key contributions of this research include:

(1) Dynamic Scheduling: The scheduler's ability to adapt task distribution based on real-time network and compute conditions is a core innovation. This flexibility significantly enhances application performance and resource utilization.

(2) Device Profiling and Predictive Profiling: Profiling devices under various conditions supports optimal task allocation, mitigating the inherent heterogeneity of edge devices. Predic-

tive profiling further refines this process, allowing the scheduler to anticipate changes and adjust task assignments proactively.

(3) Multi-level Decision-Making: The introduction of a two-tiered decision-making process minimizes scheduling overhead while maintaining decision quality, particularly under constraints of latency and local data processing.

(4) Practical Implementation: The scheduler has been implemented and evaluated across multiple scenarios, demonstrating its practicality and effectiveness in real-world applications.

Moreover, the thesis has evaluated existing middleware solutions, revealing significant gaps in current infrastructures that hinder IoT application development. By addressing these deficiencies through a novel middleware approach, this work contributes to bridging the gap between IoT device capabilities and application developer needs.

In conclusion, this thesis not only advances theoretical understanding and practical applications of edge computing but also sets a foundation for future research to build upon, particularly in optimizing distributed architectures and enhancing IoT middleware systems. The insights gained and the methodologies developed here are poised to drive further innovations in the field of edge computing, paving the way for more robust, efficient, and adaptive IoT ecosystems.

7.2 Future Work

Building upon the findings and developments presented in this thesis, several avenues for future work emerge, promising to further enhance the capabilities and efficiency of edge computing systems and distributed schedulers for IoT applications. Here are key areas where future research can make significant contributions:

Enhanced Predictive Profiling: While this thesis has laid the groundwork for predictive profiling, future efforts could focus on integrating more advanced machine learning and artificial intelligence algorithms to improve the accuracy and responsiveness of the predictive capabilities. This would allow for even more dynamic adjustments to task distributions based on predictive analytics.

Expanded Device Profiling: Expanding the scope of device profiling to include a broader range of devices and operational scenarios could further refine scheduling decisions. Future work could explore profiling under varying environmental conditions to better understand how external factors influence device performance.

Expand to More DNN Models: This algorithm is applied to DNN models with pooling layers, and we will further explore how to adapt the segmentation methods to other DNN models.

Advanced Middleware Solutions: There is substantial potential to develop more sophisticated middleware solutions that can seamlessly integrate a wider range of IoT devices and platforms. Research could focus on creating more modular and adaptable middleware architectures that facilitate easier integration and better support for cross-platform functionalities.

IoT Security and Privacy: As edge computing involves processing significant amounts of potentially sensitive data, enhancing security and privacy measures is crucial. Future work should explore the development of advanced security protocols and privacy-preserving techniques that can be integrated into the dynamic scheduling framework.

Energy Efficiency: Optimizing for energy efficiency remains a critical challenge, especially in resource-constrained edge devices. Future studies could focus on energy-efficient computing strategies that reduce power consumption without compromising performance, particularly for AI-driven tasks.

Scalability and Real-world Deployment: Scaling the proposed solutions to support larger, more complex networks and ensuring their robustness in real-world deployments is essential. Future work could involve pilot studies and large-scale implementations to validate and refine the proposed models under diverse operational conditions.

By addressing these areas, future research can significantly advance the state of the art in edge computing, making IoT systems more powerful, efficient, and capable of handling the increasing demands of modern applications.

Bibliography

[1] Wenbin Li and Matthieu Liewig. A survey of ai accelerators for edge environment. In Trends and Innovations in Information Systems and Technologies: Volume 2 8, pages 35–44. Springer, 2020.

[2] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. Bringing ai to edge: From deep learning's perspective. Neurocomputing, 485:297–320, 2022.

[3] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. IEEE internet of things journal, 3(5):637–646, 2016.

[4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. IEEE access, 8:85714–85728, 2020.

[5] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. IEEE communications surveys & tutorials, 19(4):2322–2358, 2017.

[6] Tamás Nepusz and Tamás Vicsek. Controlling edge dynamics in complex networks. Nature Physics, 8(7):568–573, 2012.

[7] Sven Akkermans, Stefanos Peros, Wouter Joosen, and Danny Hughes. Supporting iot application middleware on edge and cloud infrastructures. In Proceedings of the 10th Central European Workshop on Services and Their Composition, volume 2072. published on CEUR-WS. org, 2018.

[8] Soumya Kanti Datta, Rui Pedro Ferreira Da Costa, and Christian Bonnet. Resource discovery in internet of things: Current trends and future standardization aspects. In 2015 IEEE 2nd world forum on internet of things (WF-IoT), pages 542–547. IEEE, 2015.

[9] Padmini Gaur and Mohit P Tahiliani. Operating systems for iot devices: A critical survey. In 2015 IEEE region 10 symposium, pages 33–36. IEEE, 2015.

[10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. Journal of internet services and applications, 1:7–18, 2010.

[11] Shivaji P Mirashe and Namdeo V Kalyankar. Cloud computing. arXiv preprint arXiv:1003.4074, 2010.

[12] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to cloud computing. Cloud computing: Principles and paradigms, pages 1–41, 2011.

[13] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. The characteristics of cloud computing. In 2010 39th International Conference on Parallel Processing Workshops, pages 275–279. IEEE, 2010.

[14] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4510–4520, 2018.

[15] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and$<$ 0.5 mb model size. arXiv preprint arXiv:1602.07360, 2016.

[16] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. Proceedings of the IEEE, 107(8):1655–1674, 2019.

[17] Xuyu Wang, Xiangyu Wang, and Shiwen Mao. Rf sensing in the internet of things: A general deep learning framework. IEEE Communications Magazine, 56(9):62–67, 2018.

[18] JJ Martinez, E Diaz, J Pelechano, S Guillen, and J Martinez. Vigil system: A computer vision-based aid system. evaluation in a ring motorway section in madrid. IFAC Proceedings Volumes, 27(12):451–457, 1994.

[19] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. IEEE Computational intelligence magazine, 9(2):48–57, 2014.

[20] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In Proceedings of the 26th international conference on world wide web, pages 351–360, 2017.

[21] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: An evaluation of the state of the art. IEEE transactions on pattern analysis and machine intelligence, 34(4):743–761, 2011.

[22] William R Sherman and Alan B Craig. Understanding virtual reality. San Francisco, CA: Morgan Kauffman, 2003.

[23] Ronald T Azuma. A survey of augmented reality. Presence: teleoperators & virtual environments, 6(4):355–385, 1997.

[24] Julie Carmigniani and Borko Furht. Augmented reality: an overview. Handbook of augmented reality, pages 3–46, 2011.

[25] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. Mobile edge computing: A survey. IEEE Internet of Things Journal, 5(1):450–465, 2017.

[26] Abdulmalik Alwarafy, Khaled A Al-Thelaya, Mohamed Abdallah, Jens Schneider, and Mounir Hamdi. A survey on security and privacy issues in edge-computing-assisted internet of things. IEEE Internet of Things Journal, 8(6):4004–4022, 2020.

[27] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. Energy aware edge computing: A survey. Computer Communications, 151:556–580, 2020.

[28] Quyuan Luo, Shihong Hu, Changle Li, Guanghui Li, and Weisong Shi. Resource scheduling in edge computing: A survey. IEEE Communications Surveys & Tutorials, 23(4):2131–2165, 2021.

[29] Yiqin Deng, Zhigang Chen, Xin Yao, Shahzad Hassan, and Ali MA Ibrahim. Parallel offloading in green and sustainable mobile edge computing for delay-constrained iot system. IEEE Transactions on Vehicular Technology, 68(12):12202–12214, 2019.

[30] Nouha Kherraf, Hyame Assem Alameddine, Sanaa Sharafeddine, Chadi M Assi, and Ali Ghrayeb. Optimized provisioning of edge computing resources with heterogeneous workload in iot networks. IEEE Transactions on Network and Service Management, 16(2):459–474, 2019.

[31] Xiaofan He, Richeng Jin, and Huaiyu Dai. Peace: Privacy-preserving and cost-efficient task offloading for mobile-edge computing. IEEE Transactions on Wireless Communications, 19(3):1814–1824, 2019.

[32] 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.

[33] Qi Zhang, Lin Gui, Fen Hou, Jiacheng Chen, Shichao Zhu, and Feng Tian. Dynamic task offloading and resource allocation for mobile-edge computing in dense cloud ran. IEEE Internet of Things Journal, 7(4):3282–3299, 2020.

[34] Hossein Badri, Tayebeh Bahreini, Daniel Grosu, and Kai Yang. Energy-aware application placement in mobile edge computing: A stochastic optimization approach. IEEE Transactions on Parallel and Distributed Systems, 31(4):909–922, 2019.

[35] Xianling Meng, Wei Wang, Yitu Wang, Vincent KN Lau, and Zhaoyang Zhang. Closed-form delay-optimal computation offloading in mobile edge computing systems. IEEE Transactions on Wireless Communications, 18(10):4653–4667, 2019.

[36] Stephen Pasteris, Shiqiang Wang, Mark Herbster, and Ting He. Service placement with provable guarantees in heterogeneous edge computing systems. In IEEE INFOCOM 2019-IEEE Conference on Computer Communications, pages 514–522. IEEE, 2019.

[37] Wenyu Zhang, Zhenjiang Zhang, Sherali Zeadally, Han-Chieh Chao, and Victor CM Leung. Energy-efficient workload allocation and computation resource configuration in distributed cloud/edge computing systems with stochastic workloads. IEEE Journal on Selected Areas in Communications, 38(6):1118–1132, 2020.

[38] 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, 91:1191–1204, 2019.

[39] Defang Li, Peilin Hong, Kaiping Xue, and Jianing Pei. Virtual network function placement and resource optimization in nfv and edge computing enabled networks. Computer Networks, 152:12–24, 2019.

[40] Kai Peng, Maosheng Zhu, Yiwen Zhang, Lingxia Liu, Jie Zhang, Victor CM Leung, and Lixin Zheng. An energy-and cost-aware computation offloading method for workflow applications in mobile edge computing. EURASIP Journal on Wireless Communications and Networking, 2019:1–15, 2019.

[41] Xiaolong Xu, Hao Cao, Qingfan Geng, Xihua Liu, Fei Dai, and Chuanjian Wang. Dynamic resource provisioning for workflow scheduling under uncertainty in edge computing environment. Concurrency and Computation: Practice and Experience, 34(14):e5674, 2022.

[42] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Joel JPC Rodrigues, and Feng Xia. Deep reinforcement learning for vehicular edge computing: An intelligent offloading system. ACM Transactions on Intelligent Systems and Technology (TIST), 10(6):1–24, 2019.

[43] Haifeng Lu, Chunhua Gu, Fei Luo, Weichao Ding, and Xinping Liu. Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning. Future Generation Computer Systems, 102:847–861, 2020.

[44] Xiaoyu Qiu, Luobin Liu, Wuhui Chen, Zicong Hong, and Zibin Zheng. Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. IEEE Transactions on Vehicular Technology, 68(8):8050–8062, 2019.

[45] Shihao Shen, Yiwen Han, Xiaofei Wang, and Yan Wang. Computation offloading with multiple agents in edge-computing–supported iot. ACM Transactions on Sensor Networks (TOSN), 16(1):1–27, 2019.

[46] Jin Wang, Jia Hu, Geyong Min, Wenhan Zhan, Qiang Ni, and Nektarios Georgalas. Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning. IEEE Communications Magazine, 57(5):64–69, 2019.

[47] Samson Lasaulce and Hamidou Tembine. Game theory and learning for wireless networks: fundamentals and applications. Academic Press, 2011.

[48] Shermila Ranadheera, Setareh Maghsudi, and Ekram Hossain. Computation offloading and activation of mobile edge computing servers: A minority game. IEEE Wireless Communications Letters, 7(5):688–691, 2018.

[49] Mengyu Liu and Yuan Liu. Price-based distributed offloading for mobile-edge computing with computation capacity constraints. IEEE Wireless Communications Letters, 7(3):420–423, 2017.

[50] Jing Zhang, Weiwei Xia, Feng Yan, and Lianfeng Shen. Joint computation offloading and resource allocation optimization in heterogeneous networks with mobile edge computing. IEEE Access, 6:19324–19337, 2018.

[51] Alia Asheralieva and Dusit Niyato. Hierarchical game-theoretic and reinforcement learning framework for computational offloading in uav-enabled mobile edge computing networks with multiple service providers. IEEE Internet of Things Journal, 6(5):8753–8769, 2019.

[52] Bo Gu, Zhenyu Zhou, Shahid Mumtaz, Valerio Frascolla, and Ali Kashif Bashir. Context-aware task offloading for multi-access edge computing: Matching with externalities. In 2018 IEEE global communications conference (GLOBECOM), pages 1–6. IEEE, 2018.

[53] Quoc-Viet Pham, Tuan Leanh, Nguyen H Tran, Bang Ju Park, and Choong Seon Hong. Decentralized computation offloading and resource allocation for mobile-edge computing: A matching game approach. IEEE Access, 6:75868–75885, 2018.

[54] Bo Gu and Zhenyu Zhou. Task offloading in vehicular mobile edge computing: A matching-theoretic framework. IEEE Vehicular Technology Magazine, 14(3):100–106, 2019.

[55] Yunan Gu, Walid Saad, Mehdi Bennis, Merouane Debbah, and Zhu Han. Matching theory for future wireless networks: Fundamentals and applications. IEEE Communications Magazine, 53(5):52–59, 2015.

[56] Zhenyu Zhou, Junhao Feng, Bo Gu, Bo Ai, Shahid Mumtaz, Jonathan Rodriguez, and Mohsen Guizani. When mobile crowd sensing meets uav: Energy-efficient task assignment and route planning. IEEE Transactions on Communications, 66(11):5526–5538, 2018.

[57] Deyu Zhang, Long Tan, Ju Ren, Mohamad Khattar Awad, Shan Zhang, Yaoxue Zhang, and Peng-Jun Wan. Near-optimal and truthful online auction for computation offloading in green edge-computing systems. IEEE Transactions on Mobile Computing, 19(4):880–893, 2019.

[58] Junyi He, Di Zhang, Yuezhi Zhou, and Yaoxue Zhang. A truthful online mechanism for collaborative computation offloading in mobile edge computing. IEEE Transactions on Industrial Informatics, 16(7):4832–4841, 2019.

[59] Gang Li and Jun Cai. An online incentive mechanism for collaborative task offloading in mobile edge computing. IEEE Transactions on Wireless Communications, 19(1):624–636, 2019.

[60] Lan Li, Xiaoyong Zhang, Kaiyang Liu, Fu Jiang, Jun Peng, et al. An energy-aware task offloading mechanism in multiuser mobile-edge cloud computing. Mobile Information Systems, 2018, 2018.

[61] A-Long Jin, Wei Song, Ping Wang, Dusit Niyato, and Peijian Ju. Auction mechanisms toward efficient resource sharing for cloudlets in mobile cloud computing. IEEE Transactions on Services Computing, 9(6):895–909, 2015.

[62] Jakub Konečnỳ, Brendan McMahan, and Daniel Ramage. Federated optimization: Distributed optimization beyond the datacenter. arXiv preprint arXiv:1511.03575, 2015.

[63] Yiwen Han, Ding Li, Haotian Qi, Jianji Ren, and Xiaofei Wang. Federated learning-based computation offloading optimization in edge computing-supported internet of things. In Proceedings of the ACM Turing Celebration Conference-China, pages 1–5, 2019.

[64] Yongfeng Qian, Long Hu, Jing Chen, Xin Guan, Mohammad Mehedi Hassan, and Abdulhameed Alelaiwi. Privacy-aware service placement for mobile edge computing via federated learning. Information Sciences, 505:562–570, 2019.

[65] Yutao Jiao, Ping Wang, Dusit Niyato, and Zehui Xiong. Social welfare maximization auction in edge computing resource allocation for mobile blockchain. In 2018 IEEE international conference on communications (ICC), pages 1–6. IEEE, 2018.

[66] Nguyen Cong Luong, Zehui Xiong, Ping Wang, and Dusit Niyato. Optimal auction for edge computing resource management in mobile blockchain networks: A deep learning approach. In 2018 IEEE international conference on communications (ICC), pages 1–6. IEEE, 2018.

[67] Zehui Xiong, Yang Zhang, Dusit Niyato, Ping Wang, and Zhu Han. When mobile blockchain meets edge computing. IEEE Communications Magazine, 56(8):33–39, 2018.

[68] Xiaolong Xu, Yi Chen, Xuyun Zhang, Qingxiang Liu, Xihua Liu, and Lianyong Qi. A blockchain-based computation offloading method for edge computing in 5g networks. Software: Practice and Experience, 51(10):2015–2032, 2021.

[69] Kaile Xiao, Zhipeng Gao, Weisong Shi, Xuesong Qiu, Yang Yang, and Lanlan Rui. Edgeabc: An architecture for task offloading and resource allocation in the internet of things. Future generation computer systems, 107:498–508, 2020.

[70] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. IEEE cloud computing, 1(3):81–84, 2014.

[71] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In 2015 IEEE conference on open systems (ICOS), pages 130–135. IEEE, 2015.

[72] Roberto Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. IEEE Access, 5:8835–8850, 2017.

[73] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In 2015 3rd international conference on future internet of things and cloud, pages 379–386. IEEE, 2015.

[74] Marcel Großmann and Clemens Klug. Monitoring container services at the network edge. In 2017 29th International Teletraffic Congress (ITC 29), volume 1, pages 130–133. IEEE, 2017.

[75] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. ACM SIGARCH Computer Architecture News, 45(1):615–629, 2017.

[76] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. Ecrt: An edge computing system for real-time image-based object tracking. In Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, pages 394–395, 2018.

[77] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2348–2359, 2018.

[78] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. {COMET}: Code offload by migrating execution transparently. In 10th USENIX symposium on operating systems design and implementation (OSDI 12), pages 93–106, 2012.

[79] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In Proceedings of the 9th international conference on Mobile systems, applications, and services, pages 43–56, 2011.

[80] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In Proceedings of the sixth conference on Computer systems, pages 301–314, 2011.

[81] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In Proceedings of the 8th international conference on Mobile systems, applications, and services, pages 49–62, 2010.

[82] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. IEEE Transactions on Wireless Communications, 19(1):447–457, 2020.

[83] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In Design, Automation Test in Europe Conference Exhibition (DATE), 2017, pages 1396–1401, 2017.

[84] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2348–2359, 2018.

[85] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[86] Xiaojie Zhang, Motahare Mounesan, and Saptarshi Debroy. Effect-dnn: Energy-efficient edge framework for real-time dnn inference. In 2023 IEEE 24th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), pages 10–20, 2023.

[87] Kartikeya Bhardwaj, Ching-Yi Lin, Anderson Sartor, and Radu Marculescu. Memory- and communication-aware model compression for distributed deep learning inference on iot. ACM Trans. Embed. Comput. Syst., 18(5s), oct 2019.

[88] Yuanming Shi, Kai Yang, Tao Jiang, Jun Zhang, and Khaled B. Letaief. Communication-efficient edge ai: Algorithms and systems. IEEE Communications Surveys Tutorials, 22(4):2167–2191, 2020.

[89] Moumena A Chaqfeh and Nader Mohamed. Challenges in middleware solutions for the internet of things. In 2012 international conference on collaboration technologies and systems (CTS), pages 21–26. IEEE, 2012.

[90] Ghofrane Fersi. Middleware for internet of things: A study. In 2015 International Conference on Distributed Computing in Sensor Systems, pages 230–235. IEEE, 2015.

[91] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for internet of things: a survey. IEEE Internet of things journal, 3(1):70–95, 2015.

[92] Geoff Coulson, Gordon S Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. Distributed Computing, 15:109–126, 2002.

[93] Mauro AA Da Cruz, Joel JPC Rodrigues, Arun Kumar Sangaiah, Jalal Al-Muhtadi, and Valery Korotaev. Performance evaluation of iot middleware. Journal of Network and Computer Applications, 109:53–65, 2018.

[94] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H Campbell, and Klara Nahrstedt. Gaia: a middleware platform for active spaces. ACM SIGMOBILE Mobile Computing and Communications Review, 6(4):65–67, 2002.

[95] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In Proceedings of the 2nd international conference on Aspect-oriented software development, pages 130–139, 2003.

[96] Overview of edgex. https://www.edgexfoundry.org/.

[97] Overview of azure. https://azure.microsoft.com/en-us/services/iot-edge.

[98] Overview of kura. https://eclipse.github.io/kura/.

[99] Overview of zededa. https://zededa.com/.

[100] Overview of lfedge. https://www.lfedge.org/wp-content/uploads/2020/07/LFedge-Whitepaper.pdf.

[101] Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In Middleware 2011: ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings 12, pages 370–389. Springer, 2011.

[102] Lars Wilhelmsson, Björn Bjäre, Jonas Hansson, Chi Thu Le, and Sebastian Weber. Middleware services layer for platform system for mobile terminals, August 19 2008. US Patent 7,415,270.

[103] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 2961–2969, 2017.

[104] Mehdi Mekni and Andre Lemieux. Augmented reality: Applications, challenges and future trends. Applied computational science, 20:205–214, 2014.

[105] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, page 27–40, New York, NY, USA, 2015. Association for Computing Machinery.

[106] Overview of google brain. https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip.

[107] Di Xu, Xiang He, Tonghua Su, and Zhongjie Wang. A survey on deep neural network partition over cloud, edge and end devices. arXiv preprint arXiv:2304.10020, 2023.

[108] Sheldon Mascarenhas and Mukul Agarwal. A comparison between vgg16, vgg19 and resnet50 architecture frameworks for image classification. In 2021 International conference on disruptive technologies for multi-disciplinary research and applications (CENTCON), volume 1, pages 96–99. IEEE, 2021.

[109] Emiliano Casalicchio et al. Autonomic orchestration of containers: Problem definition and research challenges. In VALUETOOLS, 2016.

[110] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. Shimmy: Shared memory channels for high performance {Inter-Container} communication. In 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[111] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20), 2020.

[112] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In 2015 3rd international conference on future internet of things and cloud, pages 379–386. IEEE, 2015.

[113] Yan Guo, Shangguang Wang, Ao Zhou, Jinliang Xu, Jie Yuan, and Ching-Hsien Hsu. User allocation-aware edge cloud placement in mobile edge computing. Software: Practice and Experience, 50(5):489–502, 2020.

[114] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. SIGARCH Comput. Archit. News, 43(3S):27–40, jun 2015.